

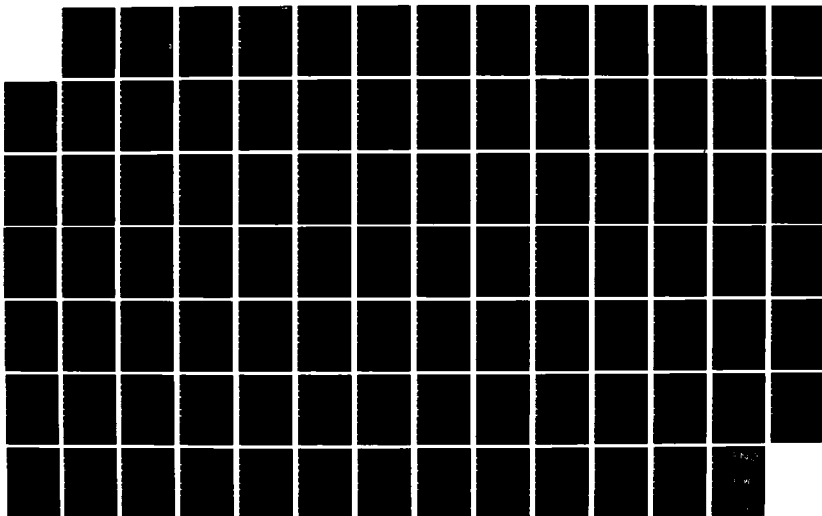
AD-A162 086

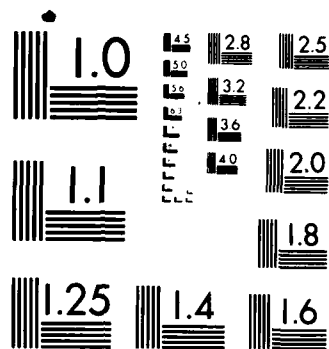
SPECIFICATION FOR MIDAS-GR MANAGEMENT OF INFORMATION  
FOR DESIGN AND ANALY (U) IOWA UNIV IOWA CITY  
APPLIED-OPTIMAL DESIGN LAB J S ARORA ET AL 03 DEC 84  
CAD-55-84 24 AFOSR-TR-85-1062 AFOSR-82-0322 F/G 9/2

1/1

UNCLASSIFIED

NL





MICROCOPY RESOLUTION TEST CHART  
NATIONAL BUREAU OF STANDARDS 1963-A

Technical Report No. CAD-SS-84.24

# Specification for MIDAS-GR: Management of Information for Design and Analysis of System: Generalized Relational Model

By

J. S. Arora and S. Mukhopadhyay

Applied-Optimal Design Laboratory  
College of Engineering  
The University of Iowa  
Iowa City, Iowa 52242

DTIC  
ELECTE  
DEC 09 1985  
S D

Prepared for the  
AIR FORCE OFFICE OF SCIENTIFIC RESEARCH  
Under Grant No. AFOSR 82-0322

Approved for public release  
distribution unlimited.

December 1984

AD-A162 086

DTIC FILE COPY

Technical Report No. CAD-SS-84.24

**SPECIFICATION FOR MIDAS/GR  
MANAGEMENT OF INFORMATION  
FOR  
DESIGN AND ANALYSIS OF  
SYSTEM: GENERALIZED RELATIONAL MODAL<sup>e</sup>**

By

J.S. Arora and S. Mukhopadhyay

**Applied-Optimal Design Laboratory  
College of Engineering  
The University of Iowa  
Iowa City, Iowa 52242**

Prepared for the  
AIR FORCE OFFICE OF SCIENTIFIC RESEARCH  
Under Grant No. AFOSR 82-0322

September 1984

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

## REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION Unclassified			1b. RESTRICTIVE MARKINGS None		
2a. SECURITY CLASSIFICATION AUTHORITY			3. DISTRIBUTION/AVAILABILITY OF REPORT Unlimited Approved for unlimited distribution		
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE			5. MONITORING ORGANIZATION REPORT NUMBER(S) AFOSR-TR-85-1002		
4. PERFORMING ORGANIZATION REPORT NUMBER(S) CAD-SS-84.24			7a. NAME OF MONITORING ORGANIZATION AFOSR/NA		
6a. NAME OF PERFORMING ORGANIZATION Applied Optimal Design Laboratory		6b. OFFICE SYMBOL (If applicable) ADL		7b. ADDRESS (City, State and ZIP Code) Bolling AFB D.C. 20332-6448	
6c. ADDRESS (City, State and ZIP Code) College of Engineering The University of Iowa Iowa City, Ia 52242		8a. NAME OF FUNDING/SPONSORING ORGANIZATION Air Force Office of Scientific Research		9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER Grant No. AFOSR-82-0322	
8b. OFFICE SYMBOL (If applicable) ABNA		8c. ADDRESS (City, State and ZIP Code) Aerospace Sciences Bolling AFB, DC 20332-6448		10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO. 61102F PROJECT NO. 2307 TASK NO. B1 WORK UNIT NO.	
11. TITLE (Include Security Classification) Specifications for MIDAS/GR: Management of Information for Design and Analysis of System: Generalized Relational Model					
12. PERSONAL AUTHOR(S) J.S. Arora and S. Mukhopadhyay					
13a. TYPE OF REPORT Interim		13b. TIME COVERED FROM 7-84 TO 12-84		14. DATE OF REPORT (Yr. Mo., Day) 1984-12-3	
15. PAGE COUNT 92		16. SUPPLEMENTARY NOTATION			
17. COSATI CODES FIELD GROUP SUB. GR.			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number) Scientific Database Management System, Generalized Relational Model, Specifications		
19. ABSTRACT (Continue on reverse if necessary and identify by block number) This report presents specification of a database management system currently under development. It is specially designed to meet the requirements of engineering applications. The stress in design lies on the flexibility of data definition capabilities, efficient management of large volume of I/O, and dynamic creation and deletion of data objects. In data definition the system takes unified approach for relational and numerical data models. It also defines a set of frequently used special data types (sparse matrix etc.) and provides uniform data structure for their efficient manipulation. The system also provides powerful data language which can be used directly from a terminal on ad hoc basis or from a host programming language. This language is non-procedural; so for a large class of problems user need not resort to branching or loops. Also, since it is precompiled instead of being interpreted at run time, it is inherently much faster. It is expected that this system will play central role in design and implementation of future systems in design optimization area.					
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED <input type="checkbox"/> SAME AS RPT. <input checked="" type="checkbox"/> DTIC USERS <input type="checkbox"/>			21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED		
22a. NAME OF RESPONSIBLE INDIVIDUAL Dr. A.K. Amos		22b. TELEPHONE NUMBER (Include Area Code) 202-767-4937		22c. OFFICE SYMBOL NA	

DD FORM 1473, 83 APR

EDITION OF 1 JAN 73 IS OBSOLETE.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE

## TABLE OF CONTENT

ABSTRACT.....	iv
LIST OF TABLES.....	v
LIST OF FIGURES.....	vi
1.0 INTRODUCTION.....	1
1.1 General.....	1
1.2 Proposed System.....	1
2.0 ARCHITECTURE.....	3
2.1 General.....	3
2.2 Data Language Interface.....	3
2.2.1 Precompiler.....	3
2.2.1.1 Optimizer.....	3
2.2.1.2 Code Generator.....	4
2.2.2 Run-time Control System.....	5
2.3 Data Storage Interface.....	5
3.0 DATA STRUCTURE.....	7
3.1 Data Types.....	7
3.1.1 Concept of Data Types.....	7
3.1.2 Primitive Data Types.....	7
3.1.3 Structured Data Types.....	8
3.1.4 Standard Data Types.....	9
3.2 Relation.....	10
4.0 DATA SUB-LANGUAGE.....	11
4.1 General.....	11
4.2 Data Manipulation Operations.....	11
4.2.1 Primitive Data Operations.....	11
4.2.2 Structured Data Operations.....	13
4.3 Data Management Operations.....	15
5.0 DISCUSSION.....	25
APPENDIX A. PRELIMINARY SPECIFICATION FOR DATABASE MANAGEMENT SYSTEM FOR ENGINEERING APPLICATIONS.....	28
A.1 Introduction.....	28
A.1.1 General.....	28
A.1.2 Proposed System.....	29
A.2 Architecture.....	30
A.2.1 General.....	30
A.2.2 Data System.....	30
A.2.3 Storage System.....	30
A.3 Data Structure.....	32
A.3.1 Base Table and Index.....	32
A.3.2 Segment.....	32
A.3.3 Field.....	33
A.4 Data Sub-Language.....	34

APPENDIX B. ARCHITECTURE FOR MIDAS/GR MANAGEMENT OF	
INFORMATION FOR DESIGN AND ANALYSIS OF SYSTEMS:	
GENERALIZED RELATIONAL MODEL.....	35
B.1 Introduction.....	35
B.2 Data Language Interface.....	40
B.2.1 General.....	40
B.2.2 Precompiler.....	40
B.2.2.1 Optimizer.....	42
B.2.2.2 Access Specification Language.....	47
B.2.2.3 Code Generator.....	55
B.2.3 Run-time Control System.....	65
B.3 Data Storage Interface.....	69
B.3.1 Area.....	69
B.3.2 Access Path.....	69
B.3.3 Concurrency Control.....	73
B.3.4 Mapping.....	74
B.3.5 Transaction Management.....	75
B.3.6 System Recovery.....	77
B.3.6.1 Soft Failure.....	77
B.3.6.2 Hard Failure.....	82
REFERENCES.....	84

## ABSTRACT

This report presents specification of a database management system currently under development. It is specially designed to meet the requirements of engineering applications. The stress in design lies on the flexibility of data definition capabilities, efficient management of large volume of I/O, and dynamic creation and deletion of data objects. In data definition the system takes unified approach for relational and numerical data models. It also defines a set of frequently used special data types (sparse matrix etc.) and provides uniform data structure for their efficient manipulation. The system also provides powerful data language which can be used directly from a terminal on ad hoc basis or from a host programming language. This language is non-procedural; so for a large class of problems user need not resort to branching or loops. Also, since it is precompiled instead of being interpreted at run time, it is inherently much faster. It is expected that this system will play central role in design and implementation of future systems in design optimization area.

Accession For	
NTIS CRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution/	
Availability Codes	
Dist	Avail and/or Special
A-1	

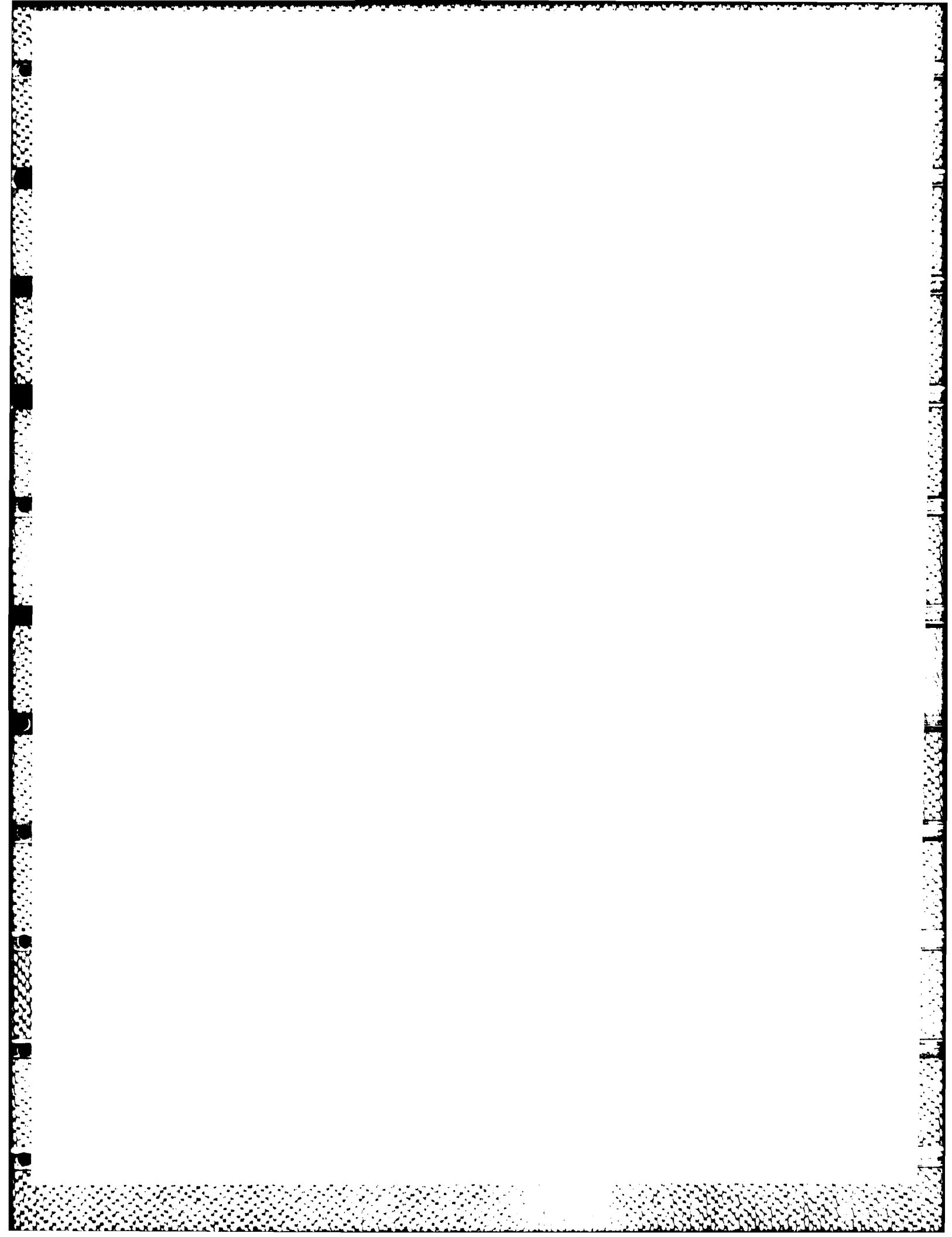


## LIST OF TABLES

Table 4.1	Functions for Primitive Data Type.....	14
Table 4.2	Functions for Matrix and Vector Data Type.....	16
Table 4.3	Functions for String Data Type.....	17

## LIST OF FIGURES

Figure 1.1	Architecture of MIDAS/GR.....	36
Figure 1.2	Structure of Data Language System DLS.....	38
Figure 1.3	Execution Step.....	39
Figure 2.1	DSI Call Structure.....	43
Figure 2.2	Scan Procedure.....	51
Figure 2.3	Scan and Build Procedure.....	51
Figure 2.4	Join using FOREACH DATA AGGREGATE.....	53
Figure 2.5	ASL Specification For Simple Query.....	56
Figure 2.6	Flow chart of the Access Routine.....	58
Figure 2.7	ASL Specification for Complex Query.....	61
Figure 2.8	Models for Implementation of Join.....	63
Figure 2.9	Combination of Models for Implementing Joins.....	64
Figure 2.10	Structure of an Access Module.....	66
Figure 3.1	Implementation of DaID.....	71
Figure 3.2	Mapping Between Pages and Slots.....	76
Figure 3.3	Database Representation (all Areas are Closed).....	79
Figure 3.4	Database Representation (A1 Modified).....	81
Figure 3.5	Save and Checkpoint Cycles.....	83



## 1. INTRODUCTION

### 1.1 General

The need for generalized database management system for engineering applications has been increasingly felt over the last decade, as the amount and complexity of data has grown. Conventional systems pose major handicaps due to the following reasons:

1. Inconsistent data storage format makes it difficult to access data collected for different applications.
2. Associating data related to the same entity is difficult when they are stored in different files.
3. The cost of storing data relating to the same entity in a number of different files (to facilitate easy access by different application programs) is excessive.
4. The inflexibility of conventional file structure and their tight binding to application programs makes future enhancement difficult.
5. Dynamic management and control of data is difficult while safeguarding its integrity, and ensuring rapid and secure access.

Objectives of the DBMS grew out of these limitations and may be summarized as follows:

1. A High-level Language. It is needed to facilitate access to stored data.
2. Control Data Integrity. Standard interface to the data sets up tight and consistent integrity control.
3. Facility of Query Language. This allows casual users to communicate with the system in english like language.
4. Model Complex Data Relations. System design should take this into consideration.

### 1.2 Proposed System

In view of the absence of a suitable database management system for scientific applications, it has been decided to develop a system to meet the requirement in this area. As stored data and its definition is extremely volatile in scientific applications, it has been felt that relational model (Codd 1970, Chamberlin 1976) will be particularly suitable. As the preliminary specification is drawn out, it is realized that conventional relational model does not represent numerical model of data (large matrix of various types) (Murthy, Reddy, Arora 1984) properly (for preliminary specification, refer to Appendix A).

Consequently in this proposed system, we define general data structure which encompasses both relational and numerical data models, and name it generalized relational model. The new system is called 'Management of Information for Design and Analysis of Systems: Generalized Relational Model (MIDAS/GR)'.

The design of MIDAS/GR is heavily indebted to the ideas used in System R. However keeping in mind its primary use (engineering applications) the stress is more on the flexibility in data definition, than the supporting data language. The system supports following basic facilities.

1. Data Independence.
2. Dynamic Data Definition.
3. Automatic Concurrency Control.
4. Flexible Authorization Mechanism.
5. Database Recovery.
6. Tuning and Usability Features.

**Data Independence.** This refers to the independence of the organization of data from the program. It has the advantage that the storage structure or access strategy may be changed without having to modify existing programs.

**Dynamic Data Definition.** Data objects or access paths may be created, modified, or dropped at any time. Therefore, it is possible to create and load just a few data objects and then begin using them immediately.

**Automatic Concurrency Control.** Locking techniques are employed to solve various synchronization problems, both at the logical level of data objects and at the physical level of pages.

**Flexible Authorization Mechanism.** The system permits users to selectively share data while retaining the ability to restrict data access. The mechanism provides protection and security, permitting information to be accessed only by properly authorized users.

**Database Recovery.** The system provides facilities for returning the database to a consistent state in case of a system crash or disk damage.

**Tuning and Usability Feature.** The system isolates the physical organization of data from logical organization. So it is possible to adjust physical data layout to improve the system performance.

## 2.0 ARCHITECTURE

### 2.1 General

The overall architecture of MIDAS/GR is described by its two main components. The lower level component is Data Storage Interface (DSI), and the upper level component is called Data Language interface (DLI).

1. Data Language Interface (DLI). This provides the external user interface, supporting tabular data structures and their operators.
2. Data Storage Interface (DSI). This provides a stored record interface to DLI.

Data language interface (DLI) consists of two components:

1. Precompiler. This is a compiler for high level language provided by the system.
2. Run-time Control System. This provides the environment for executing an application program after it has been through the precompilation process.

The precompiler in turn has two independent modules:

1. Optimizer. It decides on a strategy for implementation of each statement.
2. Code Generator. It generates machine language code to implement the chosen strategy.

For details of the system architecture, refer to Appendix B.

### 2.2 Data Language Interface

#### 2.2.1 Precompiler

The precompiler has two parts: (i) the optimizer and (ii) the code generator. These are explained in the following paragraphs.

##### 2.2.1.1 Optimizer

The object of the optimizer is to find a low cost means of executing high level statements, given data structure and available access paths. During execution of the statement the optimizer minimizes fetching pages from secondary storage into the system buffer. If necessary, the system buffer is pinned in real memory to avoid additional paging activity caused by the operating system. The cost of CPU instructions is also taken into consideration by converting the operation of data aggregate comparison to equivalent page accesses.

Since the cost of measure for the optimizer is based on disk page accesses, the physical clustering of data aggregates in the database is of great importance. Each data object may have at the most one clustering image; i.e., data aggregates near each other in a particular value ordering are stored physically near each other in the database. To understand the importance of clustering, let us suppose that we wish to scan over the data aggregates of a data object in certain order. The size of the system buffer is much less than the number of pages used to store the data object. If clustering is in different order, the location of the data aggregates will be independent of each other. In general, each data aggregate will require fetching a page from the disk. On the other hand, if the clustering order is the same, each disk page will contain several adjacent data aggregates. The number of page fetches will be reduced accordingly.

The optimizer classifies a given statement into one of the several types according to the presence of various language features, such as join, etc. Next the optimizer examines the system catalogues to find the set of indexes and links pertinent to the given statement. A rough decision procedure is then executed to find the set of 'reasonable' methods of execution. If there is more than one reasonable method, an expected cost formula is evaluated and the minimum-cost method is chosen. The parameters of cost formulas, such as data object, cardinality and number of data aggregates per page are obtained from the system catalogue.

### 2.2.1.2 Code Generator

After analyzing a high level statement, the precompiler produces an optimized code (OC) containing the parse tree and a plan for executing the statement. If the statement is a query, the OC is used to retrieve the needed data aggregates. If the statement is a view definition, the OC is stored in the form of a preoptimized code (POC). The POC can be fetched and utilized whenever an access is made via the specified view. If any change is made to the structure of a data object or to the access path (indexes and links) maintained on it, the POCs of all views defined on that data object are invalidated. Each view must be reoptimized from its defining high level code to form a new POC.

When a view is accessed via high level operators, the POC for the view can be used directly to retrieve its data aggregates. Often a query or another view definition is written in terms of an existing view. If the query or view definition is simple (e.g., a projection or restriction), it can sometimes be composed with the existing view. Their parse trees can be merged and optimized to form a new OC for the new query or view. In more complex cases, the new statement cannot be composed with the existing view definition. In these cases the POC for the existing view is treated as a formula for retrieving data aggregates. A new OC is formed for the new statement. It treats the existing view as a data object. Data aggregates can be fetched from it in only one way, i.e., by interpreting the existing POC. If the views are cascaded on other views in several levels; several levels of POCs may exist. Each level makes reference to the next.

### 2.2.2 Run-time Control System

The set of routines (machine language code) generated by the code generator, together constitutes the access module for a given program. The access module itself is stored in the database. The precompiler replaces an embedded high level statement by an ordinary host language statement to call the run-time control system.

When the object program is executed, it eventually reaches the statement calling the run-time control system which replaced the high level statement provided by DLI. Control goes to the run-time control system. It fetches access module for the program. Finally, it calls the appropriate section of the access module. This section in turn invokes various DSI operations to perform the actions required originally.

### 2.3 Data Storage Interface

This is essentially a powerful access method. Its primary function is to handle all details at the physical level. It also supports operators for data recovery, transaction management and data definition. Invocation of the DSI requires explicit use of data areas and access paths (indexes and links). It also uses the DSI generated numeric identifiers for data areas, data objects, access paths and data aggregates. The user of this system is normally the code generated by the DLI and not an application programmer. The DLI maps symbolic data object names to their internal DSI identifier.

The basic data object supported is the stored file. This is the internal representation of a data object. Rows of the data object are represented by records of the file. Records are stored in a collection of logically addressed spaces called areas. They are employed to control physical clustering. However, records need not be physically adjacent in storage.

The system depends on a user to specify the data object and the DSI access paths to be maintained on them. Access paths include indexes and links. The index-structures associate a value with one or more data aggregate identifiers (DaIDs). A DaID is an internal address allowing rapid access to a data aggregate. Indexes provide associative and sequential access on one or more fields.

Links are access paths in the DSI. They link data aggregates of one data object to data aggregates of another through pointer chains. Binary links are always maintained in a value dependent manner. The user specifies linking of data aggregates of Data objects 1 and 2 having matching values in some field(s). Ordering of data aggregates on the link is based on their value. Any attempt to define links, or to insert or update data aggregates in violation of this rule will be refused. Like an index, a link may be declared to have the clustering property. This stores each data aggregate physically near its neighbor in the link.

Each data aggregate is stored as a contiguous sequence of field values within a single page. Field lengths are also included for variable length fields. A prefix is stored with the data aggregate for use within the DSI.

The prefix contains information about the data object identifier (DoID), the pointer fields (DaIDs) for link structures, the number of stored data fields and the number of pointer fields. These numbers are employed to support dynamic creation of new fields and links to existing data objects. This is done without requiring immediate access or modification to the existing data aggregates. Data aggregates are stored only on pages reserved for data. Other pages within the area are reserved for the storage of index or internal catalogue entries. A given data page may contain data aggregates from more than one data object. In this way, extra page accesses can be avoided when data aggregates from different data objects are accessed together. A scan can be executed on a data object, rather than on index or link. Then, an internal scan is generated on all non-empty data pages within the area containing the data object. Each such data page is touched once. The prefix of each data aggregate within the page is checked to see if it belongs to the data object.

Implementation of the data aggregate identifier is a hybrid scheme. Each data aggregate identifier is a concatenation of a page number within the area, and a byte offset from the bottom of the page. The offset denotes a special entry, containing the byte location of the data aggregate in that page. This technique allows efficient utilization of space within data pages. The space is compacted and data aggregates moved with only local changes to the pointers in the slots. The slots themselves are never moved from their positions at the bottom of each data page. Thus existing DaIDs can still be used to access data aggregates. In rare cases, when a data aggregate is updated to a longer total value and insufficient space is available on its page, an overflow scheme is provided to move the data aggregate to another page. In this case, the DaID points to a tagged overflow record used to reference the other page. If the data aggregate overflows again, the original overflow record is modified to point to the newest location. Thus, a data aggregate access via a DaID almost always involves a single page access. It never involves more than two page accesses.

### 3.0 DATA STRUCTURE

#### 3.1 Data Types

##### 3.1.1 Concept of Data Types

It is customary to classify variables according to certain important characteristics. This notion of classification is particularly important in data processing, to make a choice of representation of the object in the memory of a computer.

The primary characteristic of the concept of data type is as follows:

1. It determines the set of values to which a constant belongs, or which may be assumed by a variable or an expression, or which may be generated by an operator or a function.
2. The type of a value denoted by a constant, variable, or expression may be derived from its form, or its declaration without the necessity of executing the computational process.
3. Each operator or function expects arguments of fixed type and yields a result of a fixed type.

In most cases, new data types are defined in terms of previously defined data types. Values of such a type are usually conglomerates of component values of the previously defined constituent types. They are said to be structured. If there is only one constituent type, then it is known as the base type.

##### 3.1.2 Primitive Data Types

There are five primitive types available on the system. They are as follows:

1. Integer
2. Real
3. Double precision
4. Character
5. Boolean

The type integer comprises a subset of the whole numbers whose size depends on the computer system. All operations on this type of data are exact. They correspond to ordinary laws of arithmetic. The four basic arithmetic operators are addition(+), subtraction(-), multiplication (\*), and division (div). The modulus operator is defined in terms division as

$$(m \text{ div } n) * n + (m \text{ mod } n) = m$$

Thus,  $m \text{ div } n$  is the integer quotient of  $m$  and  $n$ , and  $m \text{ mod } n$  is the associated remainder.

The type real denotes a subset of the real numbers. Arithmetic of values of type real is permitted to be inaccurate within the limits of round-off errors. This is the principle distinction between integer and real types.

The division of real numbers yielding a real valued quotient is represented by a slash (/) in contrast to div in integer division.

Double precision data is similar to real, except that twice the storage is allocated.

The type character comprises a set of printable characters. In general, the character set definition will be either ASCII or EBCDIC, depending on the computer system.

The type boolean comprises two values denoted by the identifiers true and false. the boolean operators are the logical conjunction, union, and negation. Boolean values are also yielded as a result of comparison operations.

### 3.1.3 Structured Data Types

There are four structured data types available on the system. They are as follows:

1. Record
2. Vector
3. Matrix
4. String

The type record uses the most general method to obtain structured data type. It joins elements of arbitrary, possibly themselves structured types into a compound.

Vectors and matrices are in contrast homogeneous structures. They consist of all components of the same type, called the base type. The base type in turn may be structured type. This opens up the possibility of defining a number of special data types. For example, a matrix of complex number, where complex number is a record of two real numbers.

A vector of a record may in general be treated as a relation, where each component of the vector is an occurrence of the record, and each component of the record an attribute of the relation.

If a matrix is defined as matrix of matrix, the matrix is considered made up of a number of submatrices. Matrices may be accessed row-wise, column-wise, or submatrixwise.

String is a special structured data type designed to deal with sequence of characters. String may be used as base type to define other structured data types. However, no data type except character is valid to define string. For example a 'vector of string' is quite valid; where as a 'string of vector' is erroneous.

Of the four structured data types, vector and string may be defined as variable length. Records and matrices are always of fixed length. However, they may have components which are of variable length.

### 3.1.4 Standard Data Type

The mechanism provided in the previous section allows users to define new data types. Such an approach quite naturally fits in with a design methodology based on the recognition of individual needs. Moreover such definition, at least to some extent, mirrors the resulting database structure.

Apart from the above mechanism, the system also provide a set of standard built-in data types. These data types are predefined by the system for their importance in application and are treated specially. They are as follows:

1. Relation
2. Sparse matrix
3. Upper triangular matrix
4. Lower triangular matrix
5. Banded matrix (fixed band width)
6. Banded matrix (variable band width)
7. Tri-diagonal matrix
8. Symmetric matrix

Relation is a particularly important data structure. It is defined in terms of its domains, attributes, keys, and other constraints. All standard set operations are available for such data structure.

A sparse matrix, in contrast to dense matrix, contains very few non-zero terms. The system provides efficient representation for such matrices with basic matrix operations.

Upper triangular and lower triangular matrices are the ones whose all elements below and above (respectively) the diagonal are zero. Banded matrices have non-zero element only within the band-width from the diagonal. Tri-diagonal matrix is a special type of banded matrix where band width is two. A symmetric matrix is the one which gives same matrix after transpose operation.

### 3.2 Relation

The concept of relation is based on the generalization of the relational model to suit engineering applications. A relation is essentially predefined as a vector of record, where each component of the record is defined by the user as relation attribute. Data type of an attribute may not be atomic.

Consider a relation with attributes a,b,c,d. It may be assumed to be defined as follows:

Vector of record

```

a : integer;
b : vector of integer;
c : real;
d : record
    x : integer;
    y : double precision
end
end;
```

Each relation has one or a combination of attributes uniquely identifying a tuple in the relation. The attribute (or the combination) is called primary key. An attribute with non-atomic data type can not be part of a primary key.

If more than one attribute combinations possess the unique identification property, then they are called candidate key. A candidate key that is not primary key is called an alternate key. An attribute which is primary key in some other relation is called foreign key.

The system enforces two integrity constraints:

1. Entity Integrity. No component of a primary key value may be null.
2. Referential Integrity. Any non-null value of a foreign key must correspond to its primary key value.

Unless the definition of a field is no null, any field can contain a null value. It is a special value that represents unknown or inapplicable. It has following properties.

1. Arithmetic expressions in which one of the operand is null, evaluate to null.
2. Comparisons in which one of the comparands is null, evaluate to the unknown truth value.

## 4.0 DATA SUB-LANGUAGE

### 4.1 General

This high-level language is provided for the general users to perform retrieval, update and other operations. It operates on all data objects and views. The fundamental operation of this language is mapping from physical stored record to logical data definition.

The language has following advantages:

1. **Simplicity.** Problems can be expressed more easily and concisely than in lower-level language. Simplicity in turn means productivity; i.e., ease of program development and maintenance.
2. **Completeness.** The nature of the language implies that user need not resort to loops or branching for a very large class of queries.
3. **Non-procedurality.** Such languages are frequently described as non-procedural. Its statements are high-level statement of intent on the part of the user. It means the system is able to capture the users' intent making search optimization feasible. Capturing intent is also important in authorization checking.
4. **Data Independence.** The statements include no references to explicit access paths such as indexes or physical sequence. Therefore, this language provides total physical data independence.
5. **Ease of Extension.** The power of the basic language is extended by the provision of built-in functions. The system permit users to define their own built-in functions.
6. **Support for High-level Languages.** The system is capable of supporting a variety of special purpose languages. Such languages are tailored to a particular application area, supporting its terminology and operations.

### 4.2 Data Manipulation Operations

#### 4.2.1 Primitive Data Operations

Following operations are available for integer, real, and double precision data types:

- A. Arithmetic operations
  1. Exponentiation (\*\*)
  2. Multiplication (\*)
  3. Division (/for real, div for integer)

4. Remainder (mod for integer only)
5. Addition (+)
6. Subtraction (-)

Following operations are available to all primitive data types, except boolean:

- B. Relational operations
  1. Less than (<)
  2. Less than or equal to (<=)
  3. Equal to (=)
  4. Not equal to (<>)
  5. Greater than (>)
  6. Greater than or equal to (>=)

Following operations are available to boolean data type only :

- C. Boolean operations
  1. Negation (not)
  2. Conjunction (and)
  3. Disjunction (or)
  4. Exclusive or (expr)

Also the set operator IN is available for membership test. Expressions are evaluated according to the operator precedence. Each of the operators is assigned a precedence level. They are as follows:

Precedence level	Operator
4	** not
3	* / div mod and
2	+ - or exor
1	< <= = <> > >= in

The rules of evaluation of an expression are thus:

1. If all the operators in an expression have the same precedence, the evaluation of the operations proceeds strictly from left to right (except for \*\* which proceeds strictly from right to left).

2. When operators of different precedences are present then the highest precedence operations are evaluated first (on a left to right basis), then the next highest precedence operations are evaluated.
3. Rules (1) and (2) can be over-ridden by the inclusion of parentheses in an expression. In this case those operations within the parentheses are evaluated first, with the above precedence rules being applied inside the parenthesis.

The built-in functions are listed in Table 4.1.

#### 4.2.2 Structured Data Operations

Operators and functions available for each of the structured data types are different, though vectors and matrices are treated somewhat similarly. Operators are defined for records in terms of its component types. It is possible to perform operations on a record as a whole provided all its components conform to it. For example,

```
R = record
```

```
  a : real ;
```

```
  b : integer ;
```

```
end ;
```

R <--- 5 \* R is a valid operation. However,

```
R = record
```

```
  a : real ;
```

```
  b : character
```

```
end ;
```

R <--- 5 \* R is illegal.

Most of the operations of vectors and matrices are similar. They are as follows:

##### A. Scalar operations

1. Initializing to zero
2. Initializing to 1
3. Initializing to identity matrix
4. Initializing to null (only for vector or matrix of string)
5. Multiplication or division by scalar

Table 4.1 Functions for Primitive Data Type

Definition	Generic Name	Number of Arguments
Absolute value	abs	1
Largest value	max	$\geq 1$
Smallest value	min	$\geq 1$
Square root	sqrt	1
Square	sqr	1
Exponential ( $e^{**a}$ )	exp	1
Natural logarithm	ln	1
Common logarithm (10)	log	1
Sine	sin	1
Cosine	cos	1
Tangent	tan	1
Arc sine	asin	1
Arc cosine	acos	1
Arc tangent	atan	1
Hyperbolic sine	sinh	1
Hyperbolic cosine	cosh	1
Hyperbolic tangent	tanh	1
Ord. no. to char.	chr	1
Char. to ord. no.	ord	1

6. Addition or subtraction by scalar

B. Matrix/vector operations

1. Multiplication

2. Addition and subtraction

Functions provided for matrices and vectors are listed in Table 4.2.

The system provides a variety of string operations. They are as follows:

C. String operations

1. String initialization

2. String concatenation

3. Substring designation

The string functions are listed in Table 4.3.

#### 4.3 Data Management Operations

For efficient management of data, the system supports its own language. The data language is a consistent english keyword-oriented syntax for query, as well as for data definition, data manipulation, and control. It provides facilities ranging from simple queries to complex data manipulation intended for professional programmers. The same language may be embedded in a host language program or may be used as a stand alone system. The syntax of the language is given in extended BNF notation.

However, it may be noted that a few minor ambiguities do exist; also this syntax permits the generation of some statements that are not semantically meaningful. Development of a more complete syntax is currently in progress.

In this notation, square brackets [ ] indicate optional constructs, and the braces { } indicate repeating groups of zero or more items.

```
statement ::= query
           | dml-statement
           | ddl-statement
           | control-statement

query ::= query-expr [ORDER BY ord-spec-list]

query expr ::= retrieve-clause [INTO target-list]
            FROM from-list [WHERE boolean-expr]
```

Table 4.2 Functions for Matrix and Vector Data Type

Definition	Generic Names	Number of Arguments
Transpose of matrix	trn	1
Inverse of square matrix (non-zero determinant)	inv	1
Determinant of a square matrix	det	1
Modulus of a vector	mod	1
Largest value of a matrix or vector	max	1
Smallest value of a matrix or vector	min	1
Number of elements in a vector	cnt	1
Average of a vector	avg	1
Sum of all elements of vector	sum	

**Table 4.3 Functions for String Data Type**

Definition	Generic Name	Number of arguments
Starting position of pattern in source string (source, pattern)	pos	2
Length of the string	len	1
String without leading or trailing blanks	trim	1
String with upper case character only	upc	1
String with lower case character only	lwc	1
Numerics from string	val	1
String from numerics	str	1

```

retrieve-clause ::= RETRIEVE [ UNIQUE ] ret-expr-list
                  | RETRIEVE [ UNIQUE ] *
ret-expr-list ::= ret-expr {, ret-expr }
ret-expr ::= expr | var-name.* | obj-name.*
target-list ::= var-name {, var-name }
from-list ::= obj-name [var-name] [,obj-name [var-name]]
              [, (query-expr)]
ord-spec-list ::= field-spec [ direction ] {, field-spec
                          [direction]}
field-spec ::= field-name
              | obj-name.field-name
              | var-name.field-name
direction ::= ASC | DESC
where-clause ::= WHERE boolean
boolean ::= boolean-expr | expr rel-op expr
boolean-expr ::= boolean-var | boolean-const
boolean-var ::= identifier
boolean-const ::= true | false
rel-op ::= = | <> | < | <= | >= | > | in
expr ::= term | sign term | expr add-op term
sign ::= + | -
add-op ::= + | - | or | exor
term ::= factor | term mult-op factor
mult-op ::= * | / | div | mod | and
factor ::= primary | not factor | primary ** factor
primary ::= (query-expr) | field spec | var-name | constant
           | fn-designator | (boolean)

```

fn-designator ::= fn-identifier | fn-identifier ([UNIQUE] expr)  
                   | fn-identifier (\*)

fn-identifier ::= identifier

literal ::= ( constant { , constant } )

constant ::= quoted-string | number | ROW | COL | NULL  
                   | USER | DATE

obj-name := name

image-name ::= name

link-name ::= name

name ::= [ creator. ] identifier

creator ::= identifier

user-name ::= identifier

field-name ::= identifier [( subscript-list )]

subscript-list ::= subscript { , subscript }

subscript ::= constant | var-name

var-name ::= identifier

integer ::= number

dml-statement ::= assignment

                  | insertion

                  | deletion

                  | update

assignment ::= ASSIGN TO receiver : query-expr

receiver ::= obj-name [( field-name-list )]

insertion ::= INSERT INTO receiver : insert-spec

insert-spec ::= query-expr

                  | literal

field-name-list ::= field-name { , field-name }

```

deletion ::= DELETE obj-name [ var-name ] [ where-clause ]
update   ::= UPDATE obj-name [ var-name ] set-clause-list
           [where-clause]
set-clause-list ::= set-clause { , set-clause }
set-clause ::= SET [ field-name = ] expr
              | SET [ field-name = ] ( query-expr )
ddl-statement ::= create-obj
                | create-image
                | create-link
                | define-view
                | drop
                | comment
create-obj ::= CREATE [ perm-spec ] [ shared-spec ] obj-defn
perm-spec  ::= PERMANENT | TEMPORARY
shared-spec ::= SHARED | PRIVATE
obj-defn   ::= matrix-defn | vector-defn
matrix-defn ::= MATRIX matrix-name ( integer, integer )
              [qualifier] : field-defn-list
matrix-name ::= identifier
qualifier   ::= SPARSE | UPRTRN | LWRTRN | TRIDGL | BANDED integer
              | BANDED VAR | SYMMAT
field-defn-list ::= field-defn { , field-defn }
field-defn ::= [ field-name ] ( type [, NONULL ] [, KEY ] )
type ::= primitive-type | structured-type [ : primitive-type ]
primitive-type ::= INTEGER | REAL
                  | DOUBLE PRECISION
                  | CHARACTER

```

```

        | BOOLEAN
structured-type ::= STR ( integer )
                | STR ( * )
                | VEC ( integer )
                | VEC ( * )
                | MAT ( integer, integer )
vector-defn ::= VECTOR vector-spec : field-defn-list
vector-spec ::= vector-name ( * ) | vector-name (integer)
vector-name ::= identifier
create-image ::= CREATE [ image-mod-list ] IMAGE image-name
                ON obj-name ( ord-spec-list )
image-mod-list ::= image-mod { , image-mod }
image-mod ::= UNIQUE | CLUSTERING
create-link ::= CREATE [ CLUSTERING ] LINK link-name
                FROM obj-name ( field-name-list )
                TO   obj-name ( field-name-list )
                [ ORDER BY ord-spec-list ]
define-view ::= DEFINE [ perm-spec ] VIEW obj-name
                [( field-name-list )] AS query
drop ::= DROP system-entity name
comment ::= COMMENT ON system-entity name : quoted-string
        | COMMENT ON FIELD obj-name.field-name
        : quoted-string
system-entity ::= MATRIX | VECTOR | VIEW | IMAGE | LINK
control-statement ::= grant | revoke
grant ::= GRANT [ auth ] obj-name TO user-list
        [ WITH GRANT OPTION ]

```

```
auth ::= ALL RIGHTS ON
      | operation-list ON
      | ALL BUT operation-list ON
user-list ::= user-name { , user-name } | PUBLIC
operation-list ::= operation { , operation }
operation ::= READ | INSERT | DELETE
            | UPDATE [( field-name-list )]
            | DROP | IMAGE | LINK
revoke ::= REVOKE [ operation-list ON ]
         obj-name FROM user-list
begin-trans ::= BEGIN TRANSACTION
end-trans ::= END TRANSACTION
save ::= SAVE save-point-name
restore ::= RESTORE [ save-point-name ]
```

**EXAMPLES****A. Data Definition Language**

1. CREATE TEMPORARY MATRIX STIFFNESS ( 1000, 1000 ) BANDED 20 :  
( DOUBLE PRECISION ) ;
2. CREATE PERMANENT PRIVATE VECTOR (\*) XCOMP :  
  NODE\_NO ( INTEGER , KEY ),  
  COORD ( REAL , NONULL ),  
  FORCE ( REAL ),  
  MOMENT ( REAL ) ;
3. CREATE PERMANENT SHARED VECTOR (\*) ELEMENT :  
  ELM\_NO ( INTEGER , KEY ),  
  MAT\_NO ( INTEGER ) ;
4. CREATE VECTOR (\*) NODE :  
  NODE\_NO ( INTEGER , KEY ),  
  COORD ( VEC ( 3 ) : REAL ),  
  ELM\_NO ( INTEGER , NONULL ) ;
5. CREATE UNIQUE IMAGE INODE ON NODE ( NODE\_NO ) ;
6. CREATE CLUSTERING LINK LELM  
  FROM ELEMENT ( ELM\_NO )  
  TO NODE ( ELM\_NO )  
  ORDER BY NODE\_NO ;
7. DEFINE PERMANENT VIEW YCOMP ( NNO , YCRD ) AS  
  ( RETRIEVE NODE\_NO , COORD ( 2 )  
    FROM NODE  
    WHERE ELM\_NO = 10 ) ;
8. DROP VECTOR NODE ;
9. DROP IMAGE INODE ;
10. COMMENT ON VIEW YCOMP :  
      'Limited view of node giving y-coordinate only' ;

**B. Query Language**

1. RETRIEVE \*  
  FROM STIFFNESS  
  WHERE ROW > 5  
  AND COL < 70 ;
2. RETRIEVE UNIQUE NODE\_NO , FORCE  
  FROM XCOMP  
  WHERE NODE\_NO < 50  
  AND FORCE > 5  
  ORDER BY NODE\_NO ASC ;

### C. Data Manipulation Language

1. ASSIGN TO SUBSTIFF :  
 ( RETRIEVE \*  
 FROM STIFFNESS  
 WHERE COL > 5  
 AND COL < 50 ) ;
2. ASSIGN TO SUBXCOMP ( ND , CRD , MMT ) :  
 ( RETRIEVE NODE\_NO , COORD , MOMENT  
 FROM XCOMP  
 WHERE RESTRN > 0 ) ;
3. INSERT INTO ELEMENT ( ELM\_NO , MAT\_NO ) :  
 ( RETRIEVE \*  
 FROM ELEMENT2  
 WHERE ELM\_NO > 50 ) ;
4. INSERT INTO NODE ( NODE\_NO , COORD , ELM\_NO ) :  
 ( 10 , 2.2 , 5.7 , 1.32 , 3 ) ;
5. DELETE STIFFNESS  
 WHERE ROW > 100  
 AND COL > 100 ;
6. DELETE NODE  
 WHERE NODE\_NO = 5 ;
7. DELETE ELEMENT X  
 WHERE ( RETRIEVE CNT (\*)  
 FROM NODE  
 WHERE ELM\_NO = X.ELM\_NO ) ;
8. UPDATE STIFFNESS  
 SET 25  
 WHERE ROW = 30 ;
9. UPDATE NODE  
 SET COORD = COORD \* 2.54  
 WHERE ELM\_NO IN ( 2 , 10 , 19 , 25 ) ;

### D. Data Control Language

1. GRANT READ , INSERT  
 ON NODE TO PUBLIC ;
2. GRANT ALL RIGHTS  
 ON NODE TO TOM , DICK , HARRY  
 WITH GRANT OPTION ;
3. REVOKE UPDATE , DELETE  
 ON NODE FROM DICK , HARRY ;

## 5.0 DISCUSSION

It is interesting to highlight differences between MIDAS/GR and a previous version MIDAS/R (Management of Information for Analysis and Design of System/Relational Model). These systems are briefly compared in the following.

### 1. Data Structure

a) MIDAS/GR provides integrated data definition facility, using primitive (integer, real, double precision, character) and structured (vector, matrix, string, record) data types. Relation is just one of the many user defined data types (vector of record).

MIDAS/R provides essentially a single data structure, namely relation; only the domains are extended to have non-atomic data types (vector, matrix).

b) MIDAS/GR, because of its general approach, defines matrix as easily as any other data type (say relation). As a result, matrix can be treated as a distinct entity and it can have its own composite data elements.

MIDAS/R, in contrast, defines matrix as a relation with  $n$  attributes and  $m$  occurrences. Since the maximum number of attributes in a relation (in MIDAS/R) is limited, number of component submatrices is also limited. This artificial definition not only leads to difficulty in organization of data, but also makes direct access to individual data element impossible. It is also not possible to define elements of a matrix as composite data.

c) MIDAS/GR provides several qualifier (sparse, banded, etc) to define special types of matrices frequently used in engineering applications. This allows efficient handling of storage without direct involvement from the user. Moreover, this provides uniform data structure for such matrices, effectively facilitating development and maintenance of system routines to manipulate them.

MIDAS/R essentially treats matrices as a collection of very limited number of submatrices, each of which is supposed to be dense. Since it is the users' responsibility to find efficient representation for special types of matrices, there does not exist any uniformity. As a result it is not possible to support system utilities to manipulate them, leading to multiplication of effort and consequent loss of productivity.

### 2. Data Language

a) MIDAS/GR provides unifying data language. It treats vectors (relation) and matrices similarly. It is possible to make query (or other DML operations) on individual elements of a matrix.

MIDAS/R recognizes only relations. Hence a matrix as a whole (or a limited number of submatrices) is identified as a data item. Therefore, query (or other operations) related to individual component of a matrix is not possible.

b) MIDAS/GR provides language with same syntax for both terminal users and for those using them from a programming language. This leads to ease of communication between two classes of users.

MIDAS/R uses altogether different approach for terminal interface and programming language interface.

c) MIDAS/GR provides a language which is non-procedural, i.e. its statements are statement of intent on the part of the user. This makes search optimization feasible. Also for a very large class of queries user need not resort to loops or branching. This simplicity in turn means more productivity.

MIDAS/R: Terminal interface implements relational algebra which is implicit in MIDAS/GR (consequently more compact and easy to use in MIDAS/GR). Program interface implements inefficient record at a time construct, with limited facility to define conditions. Also, update involves explicit retrieve and insert operation, which is inefficient both in terms of program construct and computer time.

d) MIDAS/GR precompiles all its statements. This reduces runtime overhead. Therefore, the system is inherently much faster.

MIDAS/R interprets its statements at runtime. Therefore, runtime overhead is high, leading to slow execution.

### 3. Data Independence

MIDAS/GR provides multiple view of same stored record. This is made possible by maintenance of multiple access paths. Apart from physical sequence, it maintains indexes and links (binary and unary) for direct access and for sequential access in a different value ordering. This leads to data independence.

MIDAS/R provides only view obtained through physical storage sequence. There is no facility to define or maintain other views or access paths (index or link). This leads to inefficient handling of data and also lack of data independence.

### 4. Concurrent Usage

MIDAS/GR allows several users to use the system concurrently, doing retrieval, update and other operations, without conflicting with other users.

MIDAS/R allows only one user to use the system at any time. This leads to inefficient use of the database.

### 5. Recovery

MIDAS/GR supports recovery in case of system failure or media crash. If the on-line storage is not destroyed (i.e., no media failure), restart is

automatic (i.e., without operator intervention). MIDAS/R does not support any form of recovery.

Apart from the major advantages in the design itself (as outlined above) MIDAS/GR is free from some vital implementation drawbacks of MIDAS/R:

1. MIDAS/R requires that every attribute name be unique in a database. This is a redundant and difficult requirement, particularly among different users.

MIDAS/GR qualifies each attribute (field) name by data object name and user name. Hence, an attribute has to be unique only within a data object (say relation).

2. MIDAS/R does not provide facility to specify whether duplicates to be removed from the result of a project operation on a relation. If it does not remove duplicates, the system will be cumbersome to use; at the same time, if it removes them all the time (even when it is not required to do so), that will entail avoidable overhead on the system.

MIDAS/GR solves this problem by providing a optional keyword 'UNIQUE' to mean that duplicates are to be removed.

3. MIDAS/R requires that every database must be explicitly defined and opened before use. This restriction is redundant for temporary databases, as they can only be used in the execution environment where they were defined.

**APPENDIX - A**

**PRELIMINARY SPECIFICATION FOR**

**DATABASE MANAGEMENT SYSTEM**

**FOR**

**ENGINEERING APPLICATIONS**

**A.1 Introduction**

**A.1.1 General**

The need for generalized database management system for engineering application is increasingly felt, as the amount and complexity of data grew over the last decade. The conventional systems are posing a major handicap due to the following reasons.

1. Inconsistent data storage format makes it difficult to access data collected for different applications.
2. Associating data, which relate to the same entity but are stored in different files, is difficult.
3. The cost of storing data relating to the same entity in a number of different files (to facilitate easy access by different application programs) is excessive.
4. The inflexibility of conventional file structure and their tight binding to application programs makes future enhancement difficult.
5. Management and controlling of data while safeguarding data integrity and ensuring rapid and secure access to data, is difficult.

The objectives of the DBMS grew out of these limitations and may be summarized as follows.

1. A high-level language to facilitate access to stored data.
2. Control data integrity. Standard interface to the data sets up tight, consistent integrity control.
3. Facility of query language. This allows casual users to communicate with the system in english like language.
4. Model complex data relationships. This can be taken advantage of in system design.

### A.1.2 Proposed System

The proposed system is based on relational model of data structure. The system distinguishes between a domain and a attribute. Attributes are drawn from a domain and represents the use of a domain within a relation.

Relations are commonly referred here as tables. There are two types of tables.

1. Base table. These tables are physically represented in the database by stored files.
2. View. These are virtual tables and does not really exists, instead derived from one or more underlying base tables. In other words, no stored file directly represent a view, instead a definition is stored in the data dictionary.

Each relation has one or a combination of attributes, that uniquely identifies a tuple in the relation. The attribute (or the combination) is called the primary key.

If more than one attribute combination possess the unique identification property, then they are called candidate key.

A candidate key that is not primary key is called alternate key.

An attribute which is primary key in some other relation is called foreign key.

A domain may optionally be designated as primary if and only if there exists some single attribute primary key defined on that domain.

The system enforces two integrity constraints.

1. Entity integrity. No component of a primary key value may be null.
2. Referential integrity. Any non-null value of a foreign key must correspond to its primary key value.

## A.2 Architecture

### A.2.1 General

The system provides following basic facilities.

1. Database recovery
2. Automatic concurrency control
3. Flexible authorization mechanism
4. Data independence
5. Dynamic database definition
6. Tuning and usability features

The system consists of two major subsystems.

1. Data system. This provides the external user interface, supporting tabular data structures and operators on these structures.
2. Storage system. This provides a stored record interface to data system.

### A.2.2 Data System

Data system consists of two components.

1. Precompiler. This is a compiler for the high-level language provided by the system. For each statement, it decides on a strategy for implementation. This process is called optimization. Having made its decision, the precompiler generates machine language routine that will implement the chosen strategy. The set of all such routines together constitutes the access module.
2. Run-time control system. This provides the environment for executing an application program that has been through the precompilation process. It fetches the access module, and then call the appropriate section of that access module, which in turn invokes various storage system operators to perform required actions.

### A.2.3 Storage System

This is essentially a powerful access method. Its primary function is to handle all details of the physical level. The user of this system is normally not a direct user, but the code generated by the data system.

The basic data object supported is the stored file i.e. the internal representation of a base table. Rows of the table are represented by records

of the file. The records need not be physically adjacent in storage. It also supports an arbitrary number of indexes over any stored file. The user of this subsystem needs to know what stored files and indexes exist and must specify the access path to be used.

## A.3 Data Structure

### A.3.1 Base Table and Index

The primary structure of the system is base table. A base table is a table that has its own independent existence. It is represented in the physical database by a stored file.

Using suitable statements, a base table may be created with appropriate fields, at appropriate segment. Just as a new base table can be created at any time, so an existing base table can be expanded at any time by adding a new column (field) at the right. The value of the new field will be null at every occurrence (the specification no null is not permitted while expanding).

It is also possible to destroy an existing base table at any time. All records in the specified base table are deleted, all indexes and views on that table are destroyed and the table itself is then also destroyed (i.e. its description is removed from the dictionary and its storage space is released).

Like base table, indexes are created and dropped using the DDL statements. However, DML statements deliberately do not include any references to the indexes. The decision as to whether to use an index or not in response to a particular data request is made not by user but by the system (the optimizing part of the precompiler).

The index may be defined by specifying major and a number of minor order (ascending/descending). For indexing purpose null values are considered to be all equal to each other and greater than any non-null value. Once created, an index is automatically maintained by the system to reflect updates on the indexed base table, till the index is dropped.

If a precompiled program has an access module that depends on the dropped index, that access module is automatically marked invalid. When that access module is next invoked, the system automatically reprepiles the original program, generating a replacement access module without using the now-vanished index. However, this process is completely hidden from the user.

### A.3.2 Segment

The database is partitioned into a set of disjoint segments. Segments provide a mechanism for controlling the allocation of storage and the sharing of data among users. Any given base table is wholly contained within a single segment; any indexes on that base table are also contained in that same segment. However, a given segment may contain several base tables and their indexes.

There are three types of segments.

1. Public segment. They contain shared data that can be simultaneously accessed by multiple users.

2. Private segment. They contain data that can be used by only one user at a time (or data that is not shared at all).
3. Temporary segment. They contain only temporary data which is lost as soon as the program terminates.

Data in public and private segment is recoverable (i.e. data will not be lost in the event of a failure), but not the one in temporary segment. This reduces the overall overhead, as the overhead associated with full support of concurrent sharing needed for public data can be avoided for private and temporary data. However, the type of a segment will be fixed at the time of system installation and can not be changed.

Each segment consists of an integral number pages. The number of pages in a given segment varies dynamically. Each segment has a predetermined maximum size (very large), but at any given time it will occupy only as much physical storage as it actually needs for the data objects it currently contains. Pages are allocated to segments as necessary and are released when the segment shrinks again.

### A.3.3 Field

Each field definition of base table includes three items.

1. Field name
2. Data type for the field
3. No null specification

The field name must be unique within the base table. The permissible data-types are as follows.

CHAR (n)	: Fixed length character string
CHAR (n) VAR	: Variable length character string
INTEGER	: Full word binary integer
REAL	: Full word floating point number
DOUBLE PRECISION	: Double word floating point number

Unless the definition of the field is no null, any field can contain a null value. It is a special value that represents unknown or inapplicable. It has following properties.

1. Arithmetic expressions in which one of the operand is null, evaluate to null.
2. Comparisons in which one of the comparands is null, evaluate to the unknown truth value.

#### A.4 Data Sub-Language

This high-level language is provided for the general users to perform retrieval, update and other operations. It operates on both base table and views. The fundamental operation of this language is mapping which is effectively horizontal subsetting and vertical subsetting of a table.

Apart from being relationally complete, this language has following advantages.

1. **Simplicity.** Problems can be expressed more easily and concisely than in lower-level language. Simplicity in turn means productivity, i.e. ease of program development and maintenance.
2. **Completeness.** Since the language is relationally complete, for a very large class of queries, the user need not resort to loops or branching.
3. **Non-procedurality.** Such languages are frequently described as non-procedural. Its statements are high-level statement of intent on the part of the user. It means the system is able to capture the users' intent, which makes search optimization feasible. Capturing intent is also important in authorization checking.
4. **Data Independence.** The statements include no references to explicit access paths such as indexes or physical sequence. Therefore, this language provides total physical data independence.
5. **Ease of Extension.** The power of the basic language is extended by the provision of built-in functions. The system permit users to define their own built-in functions.
6. **Support for High-Level Languages.** The system is capable of supporting a variety of special purpose languages; such languages being tailored to some particular application area and supporting terminology and operations specific to that area.

**APPENDIX - B**

**ARCHITECTURE FOR MIDAS/GR**

**MANAGEMENT OF INFORMATION**

**FOR**

**DESIGN AND ANALYSIS OF**

**SYSTEMS: GENERALIZED RELATIONAL MODEL**

**B.1 Introduction**

The overall architecture of MIDAS/GR is described by its two main components. The lower level component is Data Storage Interface (DSI), and the upper level component is called Data Language Interface (DLI).

1. **Data Storage Interface (DSI).** This is an internal interface which handles access to single data elements. This interface and its supporting system (Data Storage System (DSS)) is a complete storage subsystem. It manages devices, space allocation, storage buffers, transaction consistency and locking, deadlock detection, backout, transaction recovery, and system recovery. Furthermore, it maintains indexes on selected fields of relations and pointer chains across relations.
2. **Data Language Interface (DLI).** This is the external interface which can be called directly from a programming language. It may also be used to support various emulators and other interfaces. This interface and its supporting system (**Data Language system (DLS)**) provides authorization (Griffiths, Wade 1976), integrity enforcement, and support for alternative views of data. The high level data language is embedded within the DLI and is used as the basis for all data definition and manipulation. In addition, The DLI maintains the catalogs of external names (Vhrowczik 1973), since the DSI uses only system generated internal names.

Figure 1.1 (Astrahan, Blasgen, Chamberlin et. al. 1976) gives a functional view of the system including its major interfaces and components.

Data Language Interface (DLI) inturn has two independent modules:

1. **Precompiler.** This is a compiler for high level language provided by the system. The users' data statements are translated into machine language code during a preprocessing phase, and this code is stored in the database. The high level data statements in the host program are replaced with the appropriate call to the Run-time Control System (RCS).

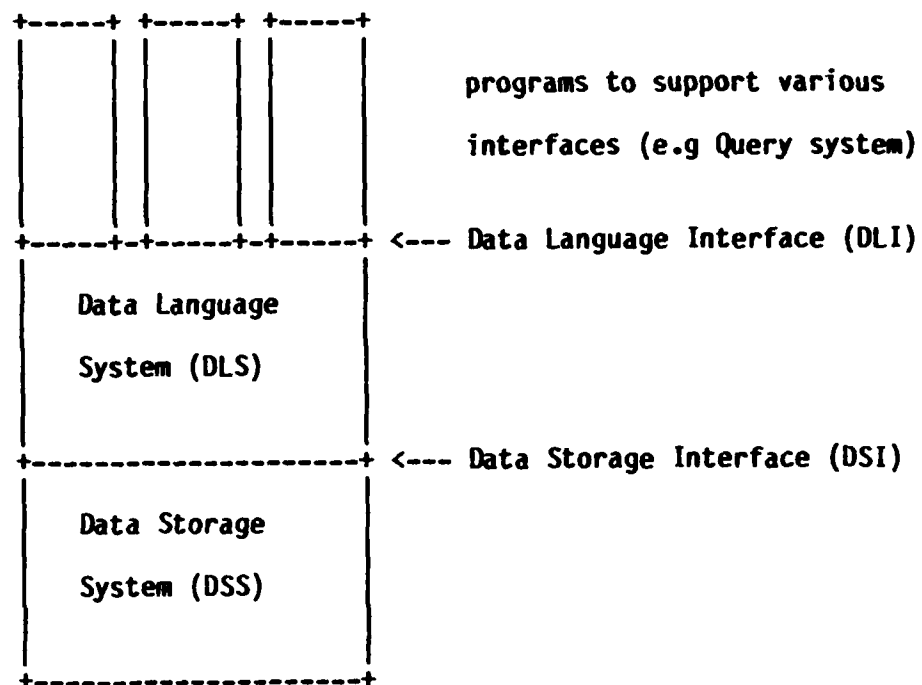


Figure 1.1 Architecture of MIDAS/GR

2. **Run-time Control System (RCS).** This provides the environment for executing an application program after it has been through the precompilation process. When the program is run, it retrieves appropriate object codes from the database and executes them.

The production of data access subroutines (object codes for each data statement) involves three steps: parsing, access path selection, and code generation. The **parser** checks the data statement for syntactic validity and translates it into a conventional parse-tree representation. The parser also returns two lists of host program variables found in data statement: a list of input variables (values to be furnished by the calling program and used in processing the statement) and a list of output variables (target locations for data to be fetched by the system). The **optimizer** uses the parse tree as input and performs following tasks.

1. Using the internal system catalog, it resolves all symbolic names in the data statement to internal database objects.
2. A check is made that the current user is authorized to perform the indicated operation on the indicated data object.
3. If the data statement operates on one or more user defined views, the definitions of the views (stored in parse-tree form) are merged with the data statement, to form a new composite parse tree which operates on real stored data objects.
4. It uses the system catalog to find the set of available indexes and other statistical information on the data object to be processed. This information is used to choose an access path and an algorithm for processing the statement.

The access strategy is specified in an internal language called the **Access Specification Language (ASL)**. It specifies completely all the access paths that may be used in the execution of a data statement. After the ASL structure has been produced by the optimizer, **code generator** produces appropriate object code to implement the specified strategy. They are called access modules. Figure 1.2 (Lorie and Nilsson 1979) shows the structure of Data Language System.

When the program is run, it makes call to RCS which inturn loads and invokes the access module for the program. The access module operates on the database by making calls to the DSI and delivers the result to the user's program. This process illustrated in Figure 1.3 (Chamberlin, Astrahan, King, et. al. 1981).

The ad hoc users are supported by a special program called the **Terminal Interface (TI)**, which controls the dialogue management and the formatting of the display terminal.

The system permits many users to be active simultaneously, performing a variety of activities. Some user may be precompiling new programs, while others are running existing programs. At the same time other users may be using the TI, querying and updating the database and creating new data objects and views. All these simultaneous activities are supported by the automatic locking subsystem built into the DSI.

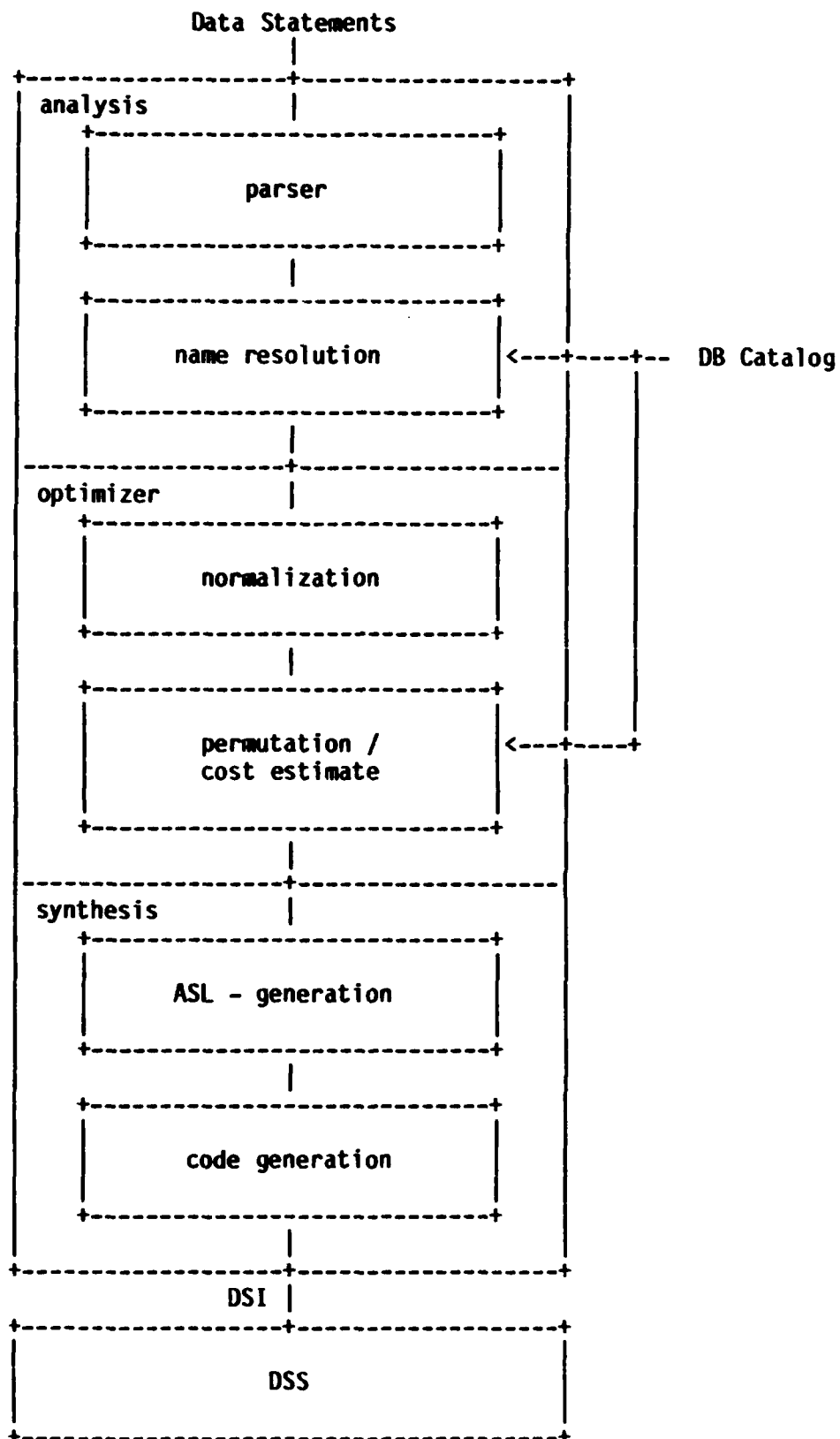


Figure 1.2 Structure of Data Language System DLS

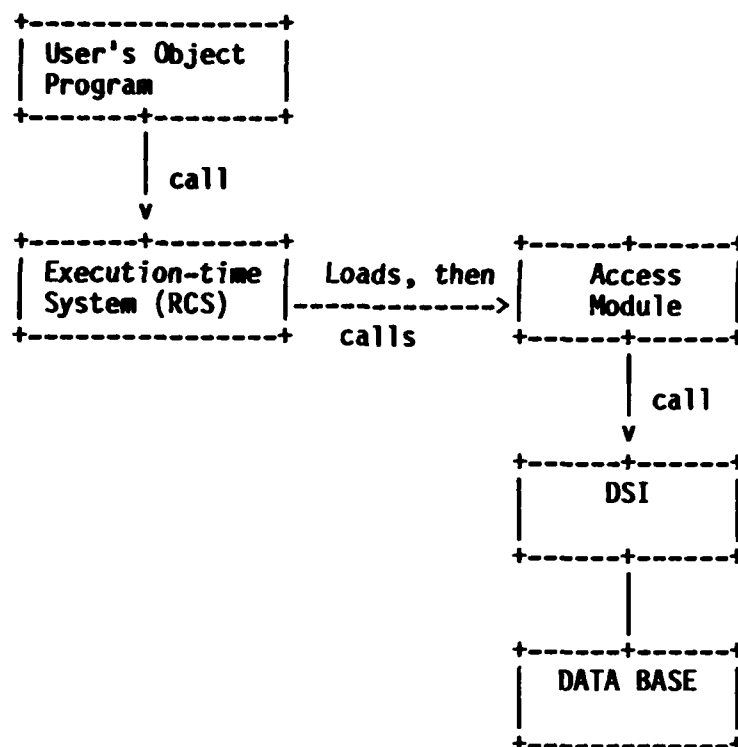


Figure 1.3 Execution Step

## B.2 Data Language Interface

### B.2.1 General

The objective of an advanced database management system (DBMS) is to provide high-level data definition, manipulation and control language, as well as a high degree of data independence. Each time the DBMS is invoked, the following operations must be performed.

1. The command string must be analyzed. This task is simple, if the command is simple and has rigid syntax. But higher the level of language, the more complex this task is. Irrespective of the level of language, the analysis often involves conversion of numeric data from external (character) format to internal (binary, decimal or floating point) format.
2. The names of the data objects (tables or file names, field or column names etc.) must be converted to their internal identifiers through use of a system catalog.
3. The DBMS must determine what processing is to be done to return data. Since, this often involves an internal search through several records, the DBMS builds an internal table using information extracted from the catalog. This table describes the required processing.
4. To actually return the data, an interpreter consults the internal table and execute a call to the underlying access method. The interpreter uses the internal table again to determine type of processing to be performed on returned data and also to determine which record to fetch next.

Steps (1), (2), and (3) are performed on every call to the DBMS; such calls may often be inside loops in application programs. So the analysis must be done repeatedly and redundantly. Step (4) may be executed several times for each DBMS call, and the level of redundancy is even higher. This may lead to severe performance degradation.

In the present system, it is decided to eliminate all execution time interpretation. For each statement, an access routine is prepared. Thus, much of the work is done at compilation time itself, rather than (repeatedly) at execution time. The compiled code for each database statement is in fact a series of calls to the DSI.

### B.2.2 Precompiler

The compilation of high-level data language consists of producing a data access routine which will invoke the access method and process the arguments and returned values as needed. This approach enables high function database language to perform as well as that available only at a lower access method level (Lorie, Wade 1979, Lang, Fernandez, Summers 1976).

The compilation of data statements is done by a preprocessor which accepts the host language-data language source code and produces two outputs. The first output is a program in host language, and the second output is an executable string (machine language data access routines). This routine is automatically stored in the database and identified by the triple (author\_name, program\_name, data statement number). At execution time DBMS fetches appropriate access routines and transfers control to it. The first output, the program in the host language may be compiled in usual way. Consider the relation

```
ELEMENT ( ELM_NO , MAT_NO , NODE_NOS )
```

Following statements may be written as part of the program.

```
LET S1 BE RETRIEVE ELM_NO , NODE_NOS
      INTO      U , V
      FROM ELEMENT
      WHERE MAT_NO = W
      AND      ELM_NO < MAX ( NODE_NOS ) ;
```

This statement is like a declaration to the database system; it associates the name S1 with the given query, but causes no processing to be performed at execution time.

The host language variables U,V,W must be declared properly. For example

```
U : Vector of integer;
V : Vector of Vector of integer;
W : integer;
```

When a statement

```
FETCH S1
```

is issued, the variables are bound.

The query will return a tuple for each ELEMENT which has a given material number (assuming material properties are defined elsewhere for each material number) and whose number does not exceed a given value.

The LET S1 ... statement is converted into a comment, since it is only a declaration to the preprocessor. Fetch statement is replaced by a group of statements.

```
INTEGER T(2)
T(1) = LOC ( U(1) )
```

```
T(2) = LOC ( V(1,1) )
```

```
CALL MIDAS ( n, LOC( T(1) ) )
```

The first argument *n* specifies the ordinal number of the data statement in the program; the second argument specifies indirectly the address of the output variables.

At execution time, the CALL MIDAS statement will invoke the DBMS, which will fetch the appropriate access routine (if it is not yet in memory) and transfer control to it.

Suppose, the ELEMENT relation is stored as one DSI relation (say relation 1005) and that an index exists on the second column (MAT\_NO), say index 2. Then the statement,

```
CALL DSI ( OPEN , SCAN_STRUCTURE , ERROR_CODE );
```

in conjunction with the structure shown in Figure 2.1 (Lorie and Wade, 1979), defines and initializes a scan of the subset of the relation identified by the condition

```
MAT_NO = 50;
```

The statement,

```
CALL DSI ( NEXT , SCAN_STRUCTURE , ERROR_CODE );
```

is then used on the scan identified by the SCAN\_STRUCTURE to fetch the successive records.

#### B.2.2.1 Optimizer

The main advantage of non-procedural data language, besides the simple data model, is that the access paths are not specified explicitly in the language. In the absence of access path specification, the compilation algorithm has to take into account the characteristics of the various access paths existing in the database (Astrahan, Blasgen, Chamberlin 1976). Therefore, given data object and available access paths the optimizer finds a low cost means of executing high level data statements. During execution of the statement the optimizer minimizes page fetching from secondary storage into the main memory buffer. If necessary, the buffer is pinned in real memory to avoid additional paging activity caused by the operating system. The cost of CPU instructions is also taken into consideration by converting operations of data element comparison to equivalent page accesses.

Since the cost of measure for the optimizer is based on disk page accesses, the physical clustering of the related data aggregates in the database is of great importance. A data object may have atmost one clustering image; i.e., data aggregates near each other in a particular value ordering are stored physically near each other in the database. To understand the importance of clustering, let us suppose that we wish to scan over a data object in certain order. The size of the system buffer is much less than the

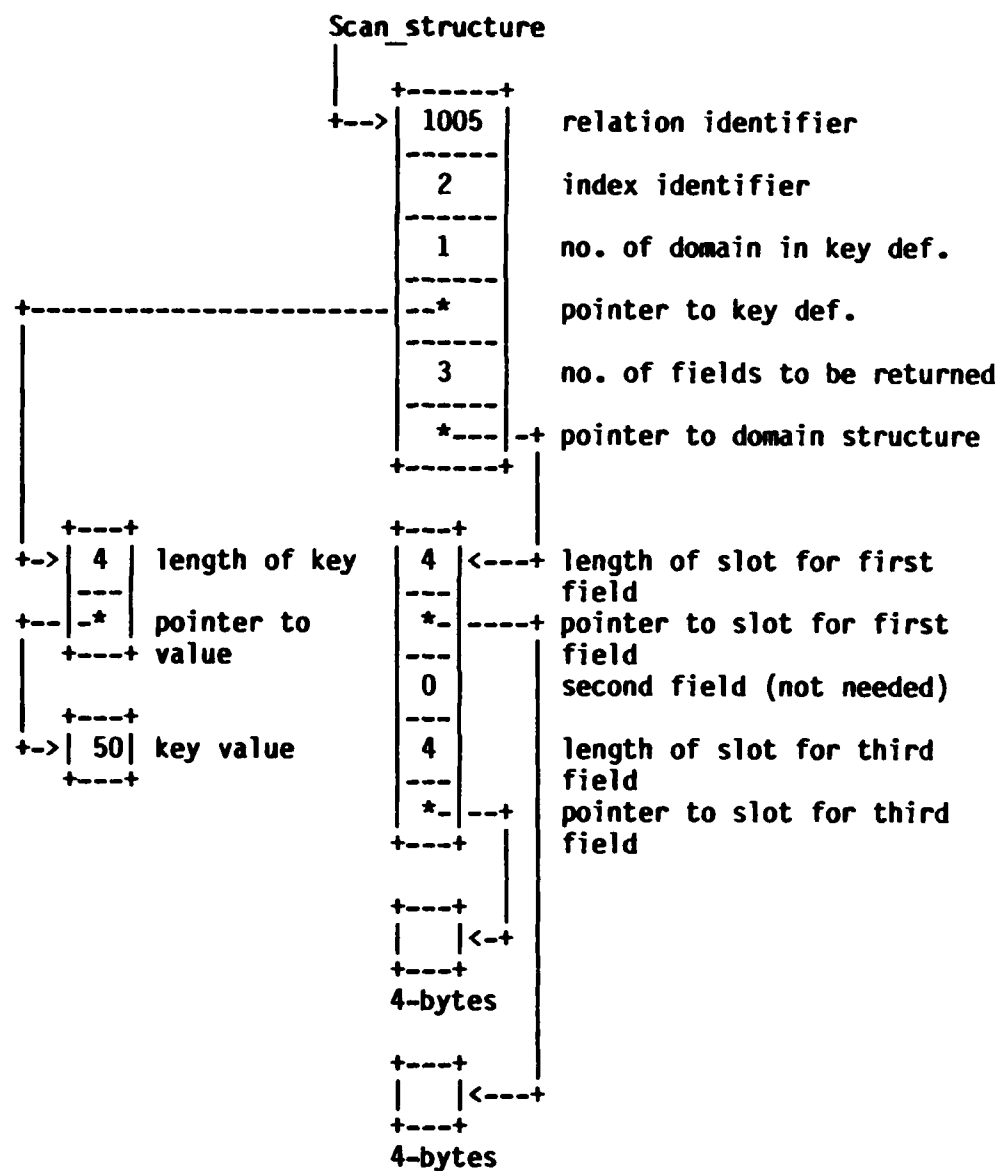


Figure 2.1 DSI Call Structure

number of pages used to store the data object. If clustering is in different order, the location of data aggregates will be independent of each other. In general, each data aggregate will require fetching a page from the disk. On the other hand, if clustering order is same, each disk page will contain several adjacent data aggregates. The number of page fetches will be reduced accordingly.

Prior to the selection of an access strategy, a 'normalization' is performed on the internal representation of the query. It involves integration of views and synonyms and conversion of some queries containing subqueries into queries containing join but no subquery. In the next phase, the optimization process takes into consideration (in principle) potentially all ASL programs which would yield the answer to the query. The one estimated to be the most efficient in terms of CPU time and database access operations is selected for synthesis.

There are two cases in the optimization process. Case 1 involves only one data object where as case 2 involves join of two (or more) data objects (Selinger, Astrahan, Chamberlin, et. al. 1979).

Let us define some useful parameters which are obtained (directly or computed) from the system catalogs.

- C Cardinality (no. of data aggregate) of the data object.
- P Number pages occupied by the data object.
- A Average number of data aggregates per data page (C/P).
- I Image (index) cardinality (no. of distinct sort field values in a given image).
- E Coefficient of CPU cost (1/E is the number of comparisons of data elements that are considered equivalent in cost to one disk page access).

An image is said to 'match' a predicate if the sort field of the image is the field which is tested by the predicate. In order for an image to match a predicate, the predicate must be a simple comparison of a field with a value.

The optimizer compares the available image with the predicates of the query to determine which of the following eight methods are available.

1. Use a clustering image which matches a predicate whose comparison operator is '='. The expected cost to retrieve all result data aggregates is  $C/(A \cdot I)$  page accesses (C/I data aggregates divided by A data aggregates per page).
2. Use a clustering image which matches a predicate whose comparison operator is not '='. Assuming half of the data aggregates satisfy the predicate, the expected cost is  $C/(2 \cdot A)$ .
3. Use a non-clustering image which matches a predicate whose comparison operator is '='. Since each data aggregate requires a page access, the expected cost is C/I.
4. Use a non-clustering image which matches a predicate whose comparison operator is not '='. Expected cost to retrieve all result data aggregate is C/2.

5. Use a clustering image which does not match any predicate. Scan the image and test each data aggregate against all predicates. Expected cost is  $(C/A) + E \cdot C \cdot N$ , where  $N$  is the number of predicates in the query.
6. Use a non-clustering image which does not match any predicate. Expected cost is  $C + E \cdot C \cdot N$ .
7. Use a scan where this data object is the only one in its segment. Test each data aggregate against all predicates. Expected cost is  $(C/A) + E \cdot C \cdot N$ .
8. Use a scan where there are other data objects sharing the segment. Cost is unknown, but greater than  $(C/A) + E \cdot C \cdot N$ , because some pages may be fetched which contain no data from the pertinent data object.

The optimizer chooses a method from this set according to the following rules:

1. If `method_1` is available, it is chosen.
2. If exactly one method among 2, 3, 5, and 7 is available, it is chosen. If more than one method is available in this class, the expected cost formulas for these methods are evaluated and the method of minimum cost is chosen.
3. If none of the above methods are available, the optimizer chooses `method_4`, if available; else `method_6`, if available; else `method_8` (Either `method_7` or `method_8` is always available for any data object).

Example of such data object is given in Section 2.2 (relation ELEMENT).

For `case_2` consider the earlier relation,

```
ELEMENT ( ELM_NO , MAT_NO , NODE_NOS )
```

and a new relation,

```
NODE ( NODE_NO , COORD , RESTRN , FORCE )
```

suppose there is a query,

```
LET S2 BE RETRIEVE ELM_NO , A
      INTO U , V
      FROM ELEMENT , ( RETRIEVE MOD (COORD)
                      INTO A
                      FROM NODE
                      WHERE RESTRN > 0 )
```

WHERE MAT\_NO = W

AND ELM\_NO < MAX (NODE\_NOS)

AND ELEMENT.NODE\_NOS = NODE.NODE\_NO ;

i.e., retrieve element number of all the elements, which has material property W and whose number is less than the highest node number on the element, and the distances (from origin) of the corresponding nodes which are restrained.

This is an instance of join type query. In most general form, it involves restriction, projection, and join. The general query has the form:

Apply a restriction on data object D, yielding D1, and apply a restriction (possibly different) to a data object E, yielding E1. Join D1 and E1 to form a new data object F, and project (if applicable) some fields from F. There are four possible methods to evaluate such query.

1. **Use Images on Key Fields.** Perform a simultaneous scan of the image on ELEMENT.ELM\_NO and the image on NODE.NODE\_NO. Advance the ELEMENT scan to obtain the next element which satisfy the predicate (MAT\_NO = W and ELM\_NO < MAX (NODE\_NOS)). Advance the NODE scan and fetch all the data aggregates whose node number matches the current node number, and which is restrained. For each such match of ELEMENT and NODE, compute distance of the node. Repeat until the image scans are completed.
2. **Sort Both Data Objects.** Scan ELEMENT and NODE using their respective clustering images, and create two files F1 and F2. F1 contains the ELM\_NO, NODE\_NOS from ELEMENT which satisfy the given predicate. F2 contains the NODE\_NO, and distance (MOD (COORD)) from NODE, which are restrained. Sort F1 and F2 on node number (This process may involve repeated passes over F1 and F2, if they are too large to fit the available main memory buffer). The resulting sorted files are scanned simultaneously and the join is performed.
3. **Multiple Passes.** ELEMENT is scanned via its clustering image and ELM\_NO, NODE\_NOS (which satisfy necessary predicates) are inserted into a main memory data structure called D. If the space in main memory is available to insert a data aggregate (say X), it is inserted. If there is no space and if X.NODE\_NOS is less than the highest NODE\_NOS value in D, the data aggregate with the highest NODE\_NOS in D is deleted and X is inserted. If there is no room and the NODE\_NOS in X is greater than the highest NODE\_NOS in D, X is discarded. After completing scan of ELEMENT, NODE is scanned via its clustering image and a data aggregate Y is obtained where the node is restrained. Then D is checked for the presence of Y.NODE\_NO. If present, the distance (MOD(Y.COORD)) is evaluated and joined to the appropriate data aggregate in D.

This process continues until all data aggregates of NODE have been examined. If any data aggregate from ELEMENT has been discarded, another scan of ELEMENT is made to form a new D, consisting of data aggregates with NODE\_NOS value greater than the current highest. NODE is scanned again and the process is repeated.

4. **DaID Algorithm.** Using the image on ELEMENT.MAT\_NO, obtain the identifiers of the data aggregates (DaID) from ELEMENT which satisfy necessary predicates. Sort them and store the DaIDs in a file F1. Do the same with NODE, using the image on NODE.RESTRN and testing for RESTRN>0, yielding DaID file F2. Perform a simultaneous scan over the images on NODE.NODE\_NO and ELEMENT.NODE\_NOS, finding the DaID pairs of data aggregates whose node numbers matches. Check each pair (DaID1, DaID2) to see if DaID1 is present in F1 and DaID2 is in F2. If they are the data aggregates are fetched and joined, and the computed distance is placed into the output.

A method can not be applied unless the appropriate access paths are available. In addition, the performance of a method depends strongly on the clustering of the data object with respect to the access paths. The optimizer chooses one of the four methods based on the cost formula.

1. There are clustering images on both ELEMENT.ELM\_NO and NODE.NODE\_NO, but no image on ELEMENT.MAT\_NO or NODE.RESTRN. In this situation, method\_1 is always chosen.
2. There are unclustered images on ELEMENT.NODE\_NOS and NODE.NODE\_NO, but no image on ELEMENT.MAT\_NO or NODE.RESTRN. In this case method\_3 is chosen if the entire working area D fits into the main memory buffer at once; otherwise, method\_2 is chosen. It may be noted that unclustered images of node number are never used in this situation.
3. There are clustering images on ELEMENT.NODE\_NOS, and NODE.NODE\_NO, and unclustered image on ELEMENT.MAT\_NO and NODE.RESTRN. In this situation, method\_4 is always chosen.
4. There are unclustered images on ELEMENT.NODE\_NOS, ELEMENT.MAT\_NO, NODE.NODE\_NO, and NODE.RESTRN. In this situation method\_3 is chosen if the entire working area D fits into the main memory buffer. Otherwise, method\_2 is chosen if more than one data aggregate per disk page expected to satisfy the restriction predicates. In the remaining cases, where the restriction predicates are very selective, method\_4 is used.

It is however important to note that image is not possible on non-atomic fields. Therefore, there will never be image on ELEMENT.NODE\_NOS. Hence all methods requiring such image is unsuitable to process this query.

#### B.2.2.2 Access Specification Language

This language allows explicit and complete specification of the access paths to be exploited in the execution of a data statement (similar to the one used in System R: Lorie, Nilsson 1979). Since, the language is intended to serve as an intermediate language in the compilation of higher level data expressions, ASL statements are represented by tree-like structures. In view of the close similarity between the syntax classes of the string language and the nodes of the tree, the language is described by an equivalent string language. Next section (Section 2.2.3 Code generation) shows examples with figures using tree structure.

Syntactically, an ASL program (in the string form) consists of procedures producing the result of the query through mutual calls.

**ASL program** ::= scanproc { ASL proc }

**ASL proc** ::= scanproc | buildproc | buildvalueproc

The ASL procedures of a program communicate solely by explicit references (generally via a call) to a procedure name and parameter/result transfer. It therefore provides a decomposition of the overall task into rather independent units.

### Query Involving Single Data Object

A scan procedure is defined as follows:

**scanproc** ::=

SCANPROC scanprocid [( param , param )];

{ temp-obj-defn; }

SCAN scanspec [ WHERE predicate ];

[ IF restriction THEN ]

RETURN ( ret-expr { , ret-expr } ) ;

END;

When the procedure is invoked for the first time, the parameters are transferred and the temporary objects that are defined in the procedure are evaluated. Thereafter, a scan is opened on a permanent database object or on one of the temporary objects just evaluated. The scan is dropped after all the data aggregates are retrieved.

**temp-obj-defn** ::=

LET RELATION relatid ( fldid { , fldid } )

= buildprocid [( id { , id } )] |

LET VECTOR vecid ( id { , id } ) = buildprocid [( id { , id } )] |

CREATE IMAGE imageid ON aggrgtid FOR fldid |

LET ( id { , id } ) = buildvalueprocid [( id { , id } )]

The statement creates a relation, vector, image, or scalar(s) respectively. The temporary objects are dropped when the instance of the procedure is terminated. For the creation of the relation or vector, the data aggregates to be inserted in the temporary objects are defined in a build procedure identified by build-procid. Scalar values are defined using buildvalue procedure.

**scanspec** ::= dtobjid { imagespec | linkspec }

A scan specification specifies the object to be scanned and the type of scan.

**imagespec** ::=

    USING IMAGE imageid [ FROM constant ] TO constant ] |

    USING IMAGE imageid AT constant { , constant }

**linkspec** ::= USING linkid ( PARENT | CHLDRN ) OF scanprocid

The link occurrence to be scanned is specified implicitly as the scan position of the newest instance of the indicated scan procedure.

The where clause is used to introduce the search predicate, which can be mapped directly on to the DSI search predicate; whereas the logical restriction of the IF statement allows for the specification of predicates which can not be expressed by an DSI search predicate.

**restriction** ::= restriction bool-op restriction |  
                  NEG restriction | pred-block

bool-op ::= AND | OR | EXOR

pred-block ::= ( { temp-obj-defn ; } comparison )

comparison ::= valcomp | setcomp

valcomp ::= arith-expr comp-op arith-expr

comp-op ::= < | > | <= | >= | = | <>

setcomp ::= ident set-op ident

set-op ::= SUBSET | PROPSUBSET | EQSET

ident ::= relid | vecid

**ret-expr** ::= arith-expr | aggr-expr

arith-expr ::= arithmetic expression

aggr-expr ::= fn-id [( arith-expr { , arith-expr } )]

The fn-id identifies functions that are provided by the language or defined by the user.

**buildproc** ::=

    BUILDPROC buildprocid [( param , param )];

    { temp-obj-defn ; }

```

SCAN scanspec [ WHERE search-predicate ];
[ IF restriction THEN ]
INSERT INTO ( RELAT | VECTOR ) : id { , id }
      [ SORTED BY id { , id } [ UNIQUE ] ];
END;

```

Syntactically the build procedure is like the scan procedure except for the INSERT clause. The semantic difference is that the build procedure does not return the individual data aggregates during a scan, but rather accumulates the whole result in an object. The DSI identifier of the object is returned to the invoker. The INSERT statement specifies the type of the object to be created (relation or vector) and the names to be given to its fields.

Figure 2.2 shows a simple graphical representation of the main ASL construct. The upper box represents the definition of temporary objects while the following ones represent the scan and the restriction part of the scan procedure. The tag (main) indicates that the scan is seen by the caller of the ASL interface.

Figure 2.3 shows a scan procedure invoking a build procedure. Note the meaning of the arrow originating in the upper box of the scan procedure, before opening the scan on A. A temporary object is constructed by invoking the build procedure. This procedure is not seen by the caller of the ASL interface and is tagged (sub) by analogy to subroutines.

### Query Involving Two or More Data Objects

Let us consider more general case of queries involving more than one data object. The FROM clause in RETRIEVE statement specifies a cartesian product subject to a restriction in the WHERE clause. Frequently, these restrictions are such that the cartesian product becomes an equi-join. Interaction among ASL procedures for the specification of joins is specified via a new ASL statement: FOREACH DATA AGGREGATE. The definition of the scanproc (and buildproc) is as follows:

```

scanproc ::=
SCANPROC ...
      ...
[ IF restriction ...]
FOREACH DATA AGGREGATE [id { , id } UNIQUE ];
[ LET id { , id } = scanprocid [( id { , id })] ; ]
RETURN ...
END;

```

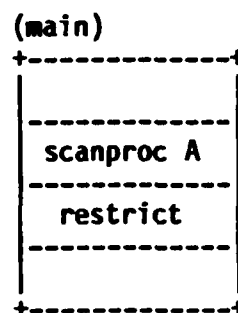


Figure 2.2 Scan Procedure

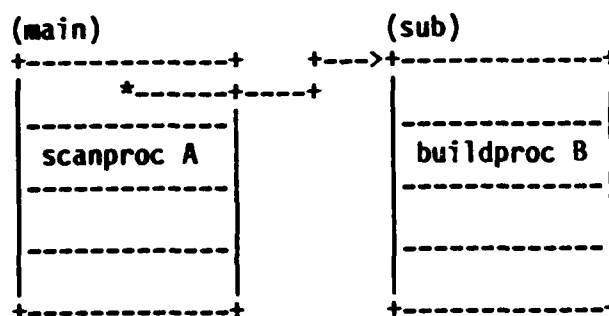


Figure 2.3 Scan and Build Procedure

Suppose a first scan procedure, say S, retrieves a data aggregate D, fulfilling the search predicate and restriction. If the FOREACH DATA AGGREGATE statement is present, the LET statement specifies a scan procedure s', which is invoked with parameter values corresponding to the current data aggregate D. The scan procedure s' returns a data aggregate D'. The fields of both D and D' may participate in the return expressions specified in the RETURN clause of s. Scan s will be advanced only after the scan s' is exhausted. For each new data aggregate in s the scan procedure s' is reinvoked with new parameter values, and the temporary objects are then reevaluated. The UNIQUE attribute can be used only if the scan is done along an access path which ensures that the data aggregates are ordered on fields specified with the UNIQUE attribute.

Figure 2.4 shows graphical representation of such join mechanism. The tag (co) is used by analogy to coroutines.

For insert, delete, and update the RETURN statement in the (main) scan procedure is replaced by a insert, delete, or update statement respectively. It specifies that the returned data aggregates must be inserted into a previously defined data object or deleted from a data object or updated in a data object.

### Examples

Consider the relations ELEMENT, XCOMP.

```
ELEMENT ( ELM_NO , MAT_NO , NODE_NO )
```

```
XCOMP ( NODE_NO , COORD , RESTRN , FORCE , MOMENT )
```

As an example of single relation query, let us consider following statement.

```
RETRIEVE ELM_NO
```

```
FROM ELEMENT
```

```
WHERE MAT_NO = 10;
```

If a relation scan is used in connection with a search predicate the ASL program becomes,

```
SCANPROC A ;
```

```
SCAN ELEMENT WHERE MAT_NO = 10 ;
```

```
RETURN ( ELM_NO ) ;
```

```
END;
```

This procedure specifies a relation scan to be opened on ELEMENT using the search predicate MAT\_NO = 10. The scan procedure A performs a OSI next operation till the scan is exhausted.

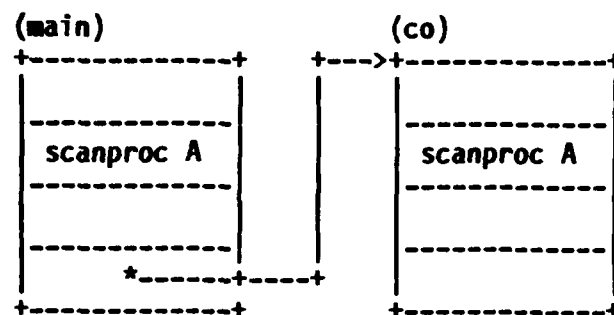


Figure 2.4 Join using FOREACH DATA AGGREGATE

Suppose that there is an image on MAT\_NO. Then the program becomes,

```
SCANPROC A ;
  SCAN ELEMENT USING IMAGE ELEMENT.MAT_NO AT 10 ;
  RETURN ( ELM_NO ) ;
END;
```

As an example of query using two relations, let us consider following statement.

```
RETRIEVE ELM_NO, COORD
FROM ELEMENT X , ( RETRIEVE NODE_NO , COORD
                  FROM XCOMP
                  WHERE RESTRN > 0 )
WHERE Y = ( RETRIEVE FORCE
          FROM XCOMP
          WHERE ELEMENT X.NODE_NOS = XCOMP.NODE_NO )
AND      SUM (Y) > MAX (Y) ;
```

i.e., retrieve element numbers and x-coordinate of its nodes (which are restrained in x-direction), from those elements which have sum of forces (in x-direction) on its nodes greater than the largest force (in x-direction) on any of its nodes.

One way of expressing the above query in ASL will be as follows:

```
SCANPROC A ;
  LET RELATION R (ND, CRD) = B ;
  SCAN ELEMENT ;
  IF ( LET Y = C (NODE_NOS) : SUM (Y) > MAX (Y) ) THEN
    FOREACH DATA AGGREGATE LET COORD = D (NODE_NOS) ;
    RETURN (ELM_NO, COORD) ;
  END ;
BUILDPROC B ;
  SCAN XCOMP WHERE RESTRN > 0 ;
```

```

INSERT INTO RELAT : NODE_NO , COORD SORTED BY NODE_NO ;

END ;

BUILDPROC C (NDNUMS) ;

SCAN XCOMP WHERE NODE_NO = ANY (NDNUMS) ;

INSERT INTO VECTOR : FORCE ;

END ;

SCANPROC D (NDNUMS) ;

SCAN R WHERE ND = ANY (NDNUMS) ;

RETURN (COORD) ;

END ;

```

It is clear that associated with every RETRIEVE statement, there can be a number of ASL programs, depending on the availability of access paths. Assuming that the optimizer part of the compiler is able to make suitable cost estimation, the generation of ASL program is straight forward.

### B.2.2.3 Code Generator

The code generator translates the ASL structure produced by the optimizer into a machine language routine, which implements the chosen access path. Compilation of data language consists of producing data access routines which will invoke access method and process the arguments and returned values as needed. The subroutine consists almost entirely of instructions needed to invoke the access method (Lorie & Wade, 1979).

Consider the relation XCOMP,

```
XCOMP ( NODE_NO , COORD , RESTRN , FORCE , MOMENT )
```

and a simple query,

```

RETRIEVE COORD , ( COORD + RESTRN )

FROM XCOMP

WHERE RESTRN < 50

AND FORCE < MOMENT ;

```

Suppose the optimizer produces the ASL structure as shown in Figure 2.5. The predicate `restrn<50` is omitted because it is incorporated into the access method. The DSI will ensure that any data aggregate it returns meets that test.

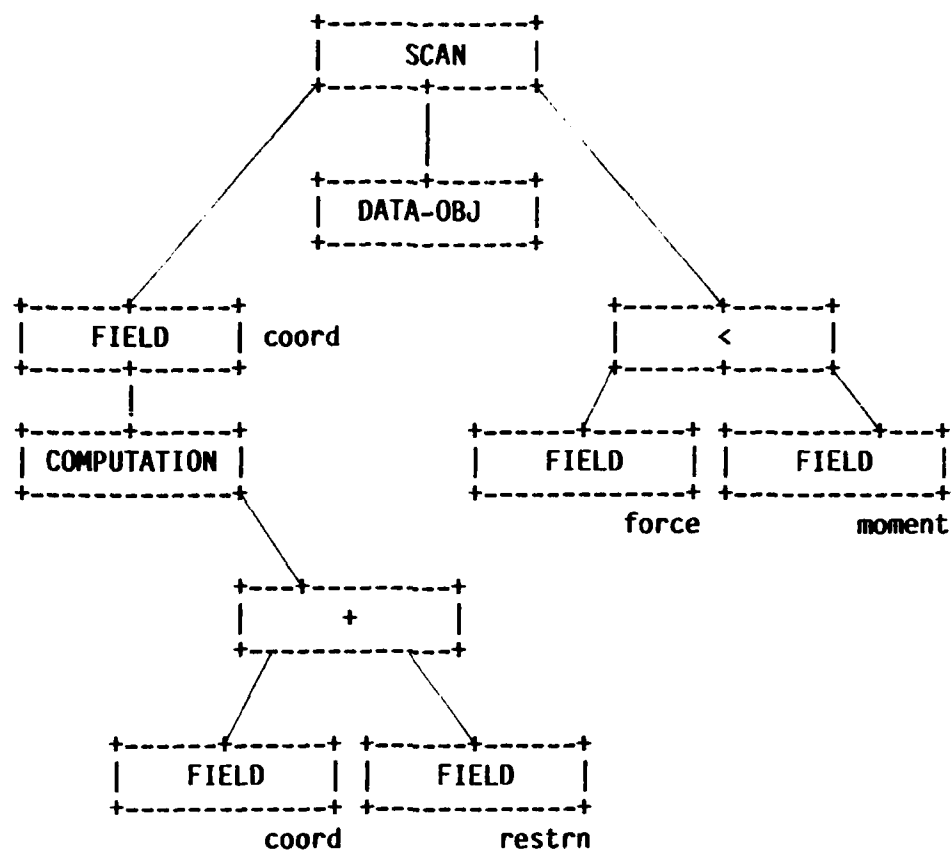


Figure 2.5 ASL Specification For Simple Query

Figure 2.6 (Lorie & Wade 1979) shows the flowchart of the access routine, which should be written to evaluate such query. Box 0 establishes the normal subroutine linkage. Box 1 tests the type of the call. If it is a call for open, the input variable is bound in box 2. Then in box 3 the DSI is invoked to open the scan. If it is not an open call, box 4 invokes the DSI to fetch the next record. Once the WHERE clause is evaluated (box 5), box 6 tests the result; if it is false, the code branches back to get the next data aggregate from DSI. If the fetched data aggregate satisfies the WHERE clause, the computation of the values to be returned is performed in box 7. In box 8 the obtained values returned to the program variables. Box 9 contains all the exits, normal and abnormal.

Boxes 1, 3, 4, 6, and 9, together forms the basic process. The skeleton of a basic process is called a **model**. The pieces of code in individual boxes in the flowchart are called **fragments**. These fragments are written as assembler routines. They are never executed in place, but are relocated when used by a particular model.

The code generator uses a general tree-walking algorithm. When the SCAN node is encountered, it is placed on the top of the stack, and a semantic routine (corresponding to 'place SCAN on stack') is invoked. This routine produces fragments corresponding to boxes 0 and 1, and generates instructions for copying data values from user's program to pre-allocated storage (slots) in the data area. It then produces fragments corresponding to boxes 3, 4, and 5. The tree-walk continues with the right branch of the SCAN node; and code is generated which evaluates WHERE clause. When the right branch is fully traversed, the nodes of the right branch is popped off the stack, and the SCAN node is again at the top of the stack. However, the SCAN node is not removed, instead a semantic routine is called to produce the fragments corresponding to box 6, and then the left branch of the SCAN node is traversed. During the processing of the left branch, code is generated to compute the values in the RETRIEVE-list and to return them to the user's program. When the left branch is fully traversed, the SCAN node reappears at the top of the stack; the fragments for box 9 is produced and the SCAN node is popped off the stack. It may be noted that some fragments branch to labels in other fragments. These external references are resolved using symbol table.

The main characteristics of this model (**model 1**) is that it contains an internal loop; inside the loop an action is applied to every data aggregate returned by the SCAN. However, other models are needed to support the following families of queries. **Model 2** is used to generate routines which compute a function (MAX, MIN, AVG, CNT etc.) on a set of data elements.

Two models described above compile queries that use a single data object. In both the cases a single routine is produced by the code generator.

For complex queries (queries involving join), consider the relations ELEMENT, and MATERIAL,

```
ELEMENT ( ELM_NO , MAT_NO , NODE_NOS )
```

```
MATERIAL ( MAT_NO , ELM_TYP , MAT_PROP )
```

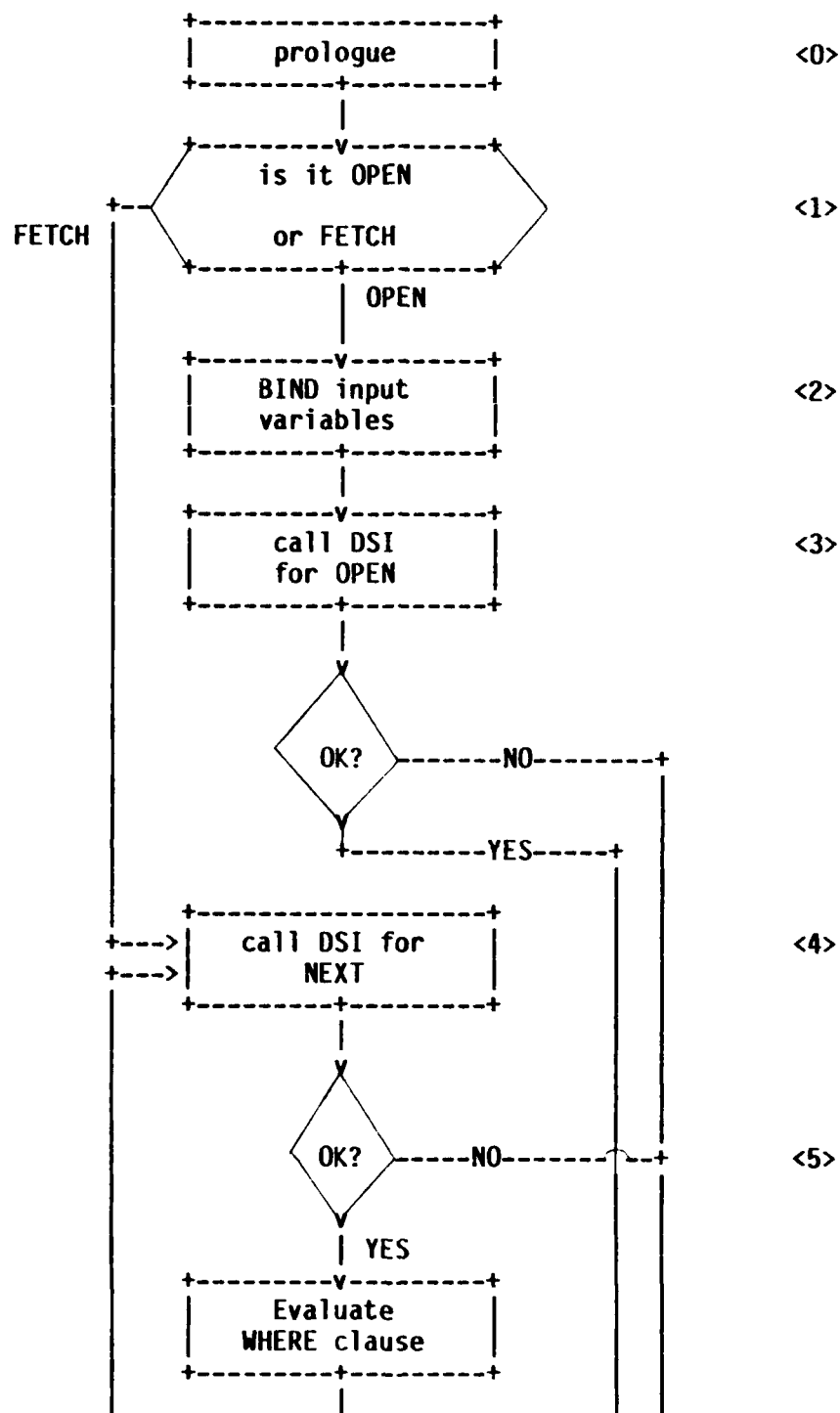


Figure 2.6 Flowchart of the Access Routine (continued on next page)

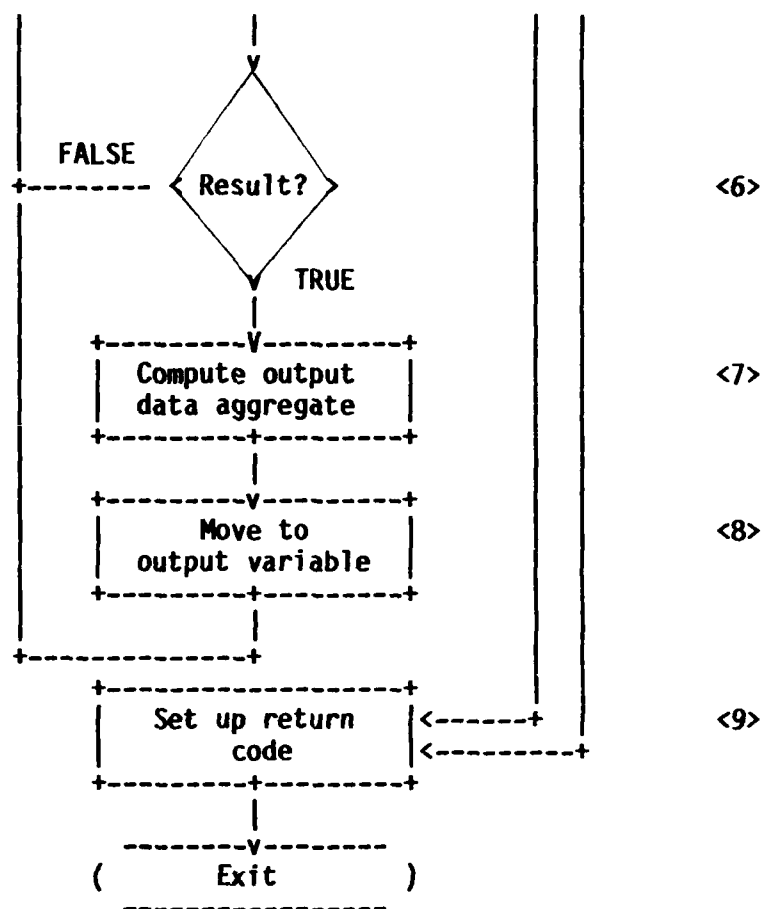


Figure 2.6 Flowchart of the Access Routine (continued from previous page)

and the query,

```

RETRIEVE ELM_NO , MAT_PROP
FROM ELEMENT X , MATERIAL Y
WHERE X.MAT_NO = Y.MAT_NO
AND ELM_TYP < ( RETRIEVE ELM_TYP
                FROM MATERIAL
                WHERE MAT_NO = 7 )
AND ELM_NO > ( RETRIEVE AVG (ELM_NO)
              FROM ELEMENT
              WHERE MAT_NO = X.MAT_NO ) ;

```

i.e., retrieve element number and material property of those elements, so that the element type is less than the element type of material number 7 and element number is greater than average element number with corresponding material number.

A possible strategy is computing the value, say  $a$ , of the first subquery before initiating the scan on ELEMENT; then for each data aggregate in the scan the first predicate can be handled as if it were  $ELM\_TYP < a$ . However, the second predicate refers to the value of a subquery which is correlated to the outer-level query ( $X.MAT\_NO$ ), and therefore, that subquery must be evaluated for every data aggregate that satisfies the first predicate. For every data aggregate that passes through both the predicates, the matching data aggregate in MATERIAL must be retrieved by using, for example, a scan along the index on the field MAT\_NO.

The ASL representation for such a strategy is shown in Figure 2.7 (Lorie, Wade 1979). The subtree<sub>(1)</sub> defines the main scan on the relation ELEMENT. A list of nodes can appear between the QUERY and DUMMY nodes. They define the actions to be taken at open time of the scan. Here a single action is to be performed which consists of computing a unique value  $a$ . The subtree<sub>(3)</sub> defines the strategy to be used to compute  $a$ . In subtree<sub>(1)</sub> the first predicate uses  $a$  as a constant.

The second predicate in subtree<sub>(1)</sub> contains a pointer to a QUERY node (subtree<sub>(4)</sub>). This construct indicates that the value of the subquery can not be computed once at the beginning of the scan; but needs to be evaluated for every data aggregate in the scan on ELEMENT.

The DUMMY node points also to a FOREACH node, which itself points to SCAN<sub>2</sub>. This construct is used to specify how a join is computed. In this case, for each data aggregate  $D$  in the scan on ELEMENT (SCAN<sub>1</sub>), the scan in subtree<sub>(2)</sub> is executed. It returns one MATERIAL data aggregate  $D'$  at a time. This  $D'$  concatenated with  $D$ , provide a data aggregate of the join. It may be noted that in SCAN<sub>2</sub>, the fields from both data aggregates  $D$  and  $D'$  are available.

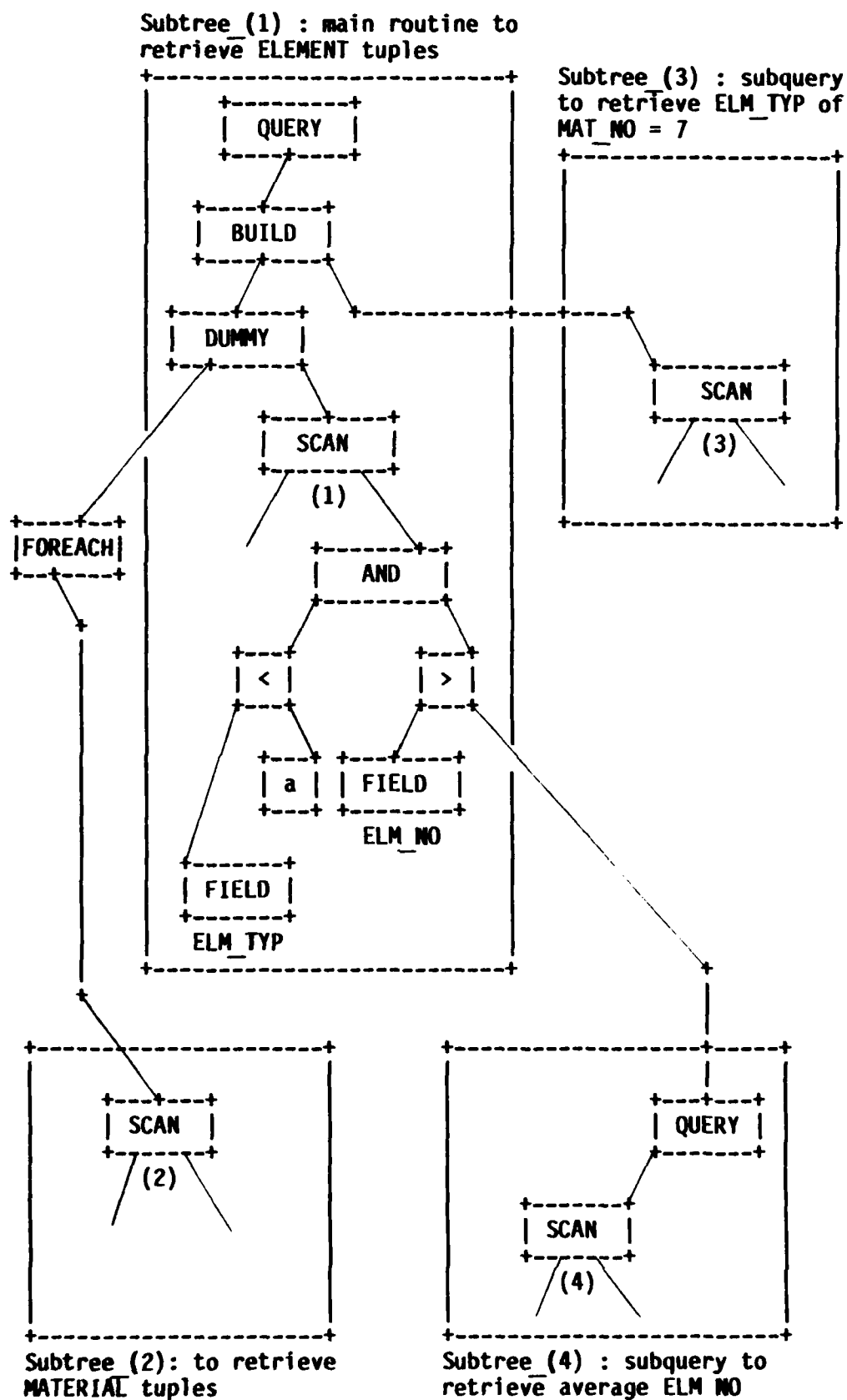


Figure 2.7 ASL Specification for Complex Query

The ASL constructs SCAN, DO AT OPEN, SUBQUERY, and FOREACH can be used to specify arbitrarily complex strategies. ASL decomposes the complex strategy into simple, virtually independent blocks, each of which has the same complexity as the ASL structure used in Figure 2.5. However, we will require different models to compile such strategies.

SCAN<sub>1</sub> implements essentially the same function as model<sub>1</sub> except that it requires a slightly more complex model because of join. Model<sub>3</sub> knows how to call the routine for SCAN<sub>2</sub> to implement the join.

The routine corresponding to SCAN<sub>2</sub> must know that it is being called by model<sub>3</sub> routine. Although its structure is similar to model<sub>1</sub> routine, model<sub>4</sub> is introduced. Model<sub>4</sub> assumes that the join is between two objects only. In a join of several objects model<sub>5</sub> is used; it is called by model<sub>3</sub> routine, but needs to call a routine to implement the next leg of the join. SCAN<sub>3</sub> uses model<sub>1</sub> to return its value. SCAN<sub>4</sub> uses model<sub>2</sub> to return the value of a function. Figure 2.8 shows the models used to implement joins.

The code generation proceeds as follows: The first QUERY node identifies the beginning of the subtree corresponding to the main routine. The list of nodes between the QUERY node and the DUMMY node is analyzed. For each subquery rooted at these nodes a block of virtual memory is reserved, and an entry is made in the SUBQUERY TABLE. This block of memory subsequently receives the corresponding generated code. Each such entry consists of a pointer to the reserved memory and a pointer to the QUERY node. Next, the SCAN node following the FOREACH node (SCAN<sub>2</sub>) is traversed. Its corresponding routine is therefore generated first. The second SCAN node to be traversed is SCAN<sub>1</sub> and the main routine is therefore generated next. During this process the fragments for model<sub>3</sub> are output. The instructions for calling the subroutine corresponding to SCAN<sub>3</sub> are inserted before the fragment for opening SCAN<sub>1</sub> is copied into the output area. Similarly, the instructions to call the subroutine corresponding to SCAN<sub>4</sub> are introduced at the appropriate place, during compilation of the WHERE clause. Also a block of storage is reserved, and an entry is made in the SUBQUERY TABLE. The address of the area is used for calling the routine even though the called subroutine has not yet been generated.

When the generation of the main routine is completed the SUBQUERY TABLE is analyzed to see if it still contains a subquery to be processed; if it does, the entry is removed from the table and the corresponding code is generated. Code generation process is repeated until no entry remains in the SUBQUERY TABLE.

There is one more way to join two data objects. Suppose, the data aggregates in both data objects are ordered on join field(s), then a merge-like scan is used to find the matching data aggregates. Controlling this process calls for specific models. Model<sub>6</sub>, 7, and 8 are substituted for models<sub>3</sub>, 4, and 5. In a join of more than two objects, each join can employ either method. To support all combinations, two more models are introduced. Model<sub>9</sub> and 10 understand how to be called by a routine for implementing a method<sub>1</sub> (method<sub>2</sub>) join and how to call a routine for implementing a method<sub>2</sub> (method<sub>1</sub>) join. Figure 2.9 illustrates some possible combinations of the models. The number on each arrow refers to the join method used for that particular 'leg' of the join.

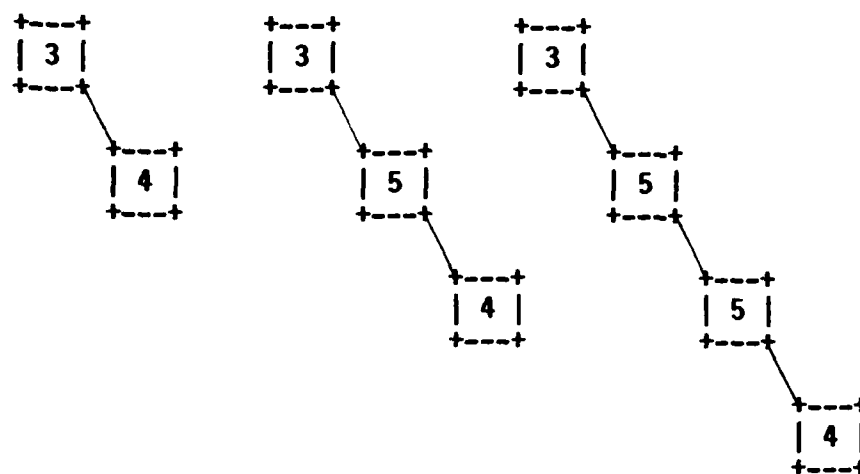


Figure 2.8 Models for Implementation of Join

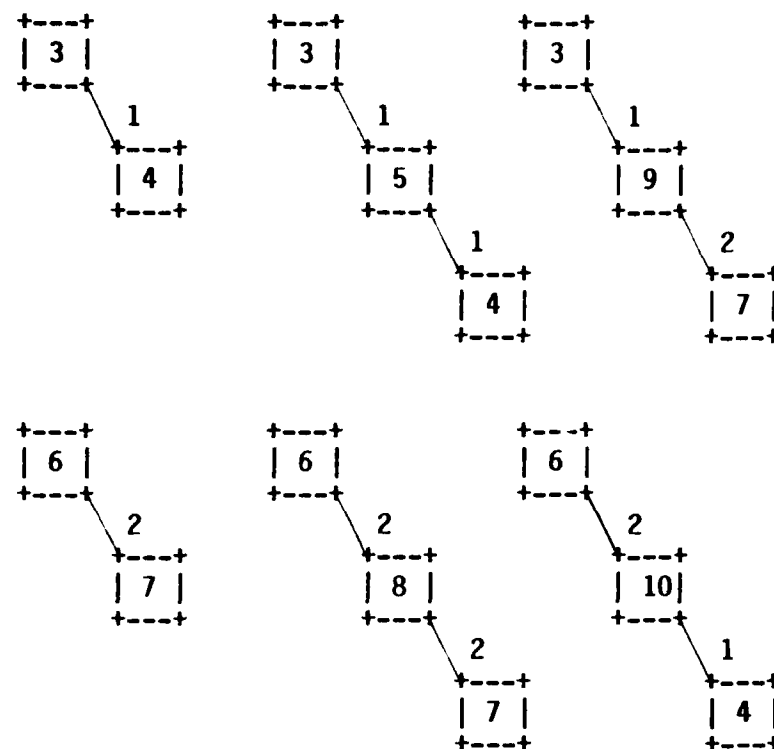


Figure 2.9 Combination of Models for Implementing Joins

These access routines are called **sections**. After all the statements in a program have been translated into sections, the sections are collected together to form an **access module**. In the header of the access module is placed a **Section Location Table** which lists the relative byte offset of each section within the access module. Each section has a **Relocation Directory** which lists the offsets within the section of all internal pointers which must be relocated before the section can be used. In addition to machine language code, each section holds the parse-tree structure of the original data statement. The structure of the access module is shown in Figure 2.10.

The data access subroutines generated, relies on the existence of particular access paths. Therefore, if that access path is deleted from the database, the access routine becomes invalid. The system includes dependency cross reference mechanism which forces the automatic regeneration of a data access routine which depended on a deleted access path. Since the parse tree structure is also stored in the database, only the access path selection and code generation phases must be redone. In no case is it necessary to recompile the application program itself. The entire process is totally transparent to the end user.

### B.2.3 Run-time Control System (RCS)

The system supports two different types of programming against a database.

1. Ad hoc queries and updates, which are usually executed only once, and
2. Canned programs, which are installed in a program library and executed hundreds of times.

The same data language is available in both these environments. These features include statements to query and update a database, to define and delete database objects such as matrix, vector, relation and views, and indexes, and to control access to the database by various users.

When a user invokes a program which has been precompiled, the normal facilities of the operating system are used to load and start the object program. MIDAS becomes aware of the program, when it makes its first call to RCS. On the first such call, RCS checks the authority of the current user to invoke the indicated access module, and checks that the access module is still valid. If these checks are successful, the access module is loaded from the database into virtual memory. Its internal pointers are adjusted using the relocation directory of each section. Then the control is passed to the indicated section. On subsequent calls to the same access module, the authorization check, loading and relocation steps are bypassed, and control passes directly to the indicated section. The machine language code in the section examines the operation code of the call (e.g. OPEN or FETCH) and process the original statement from which it was compiled, using as needed the host program variables which were passed with the call.

Since all name binding, authorization checking, an access path selection are done during the precompilation step, the resulting access module is dependent on the continued existence of the data objects it operates on, the

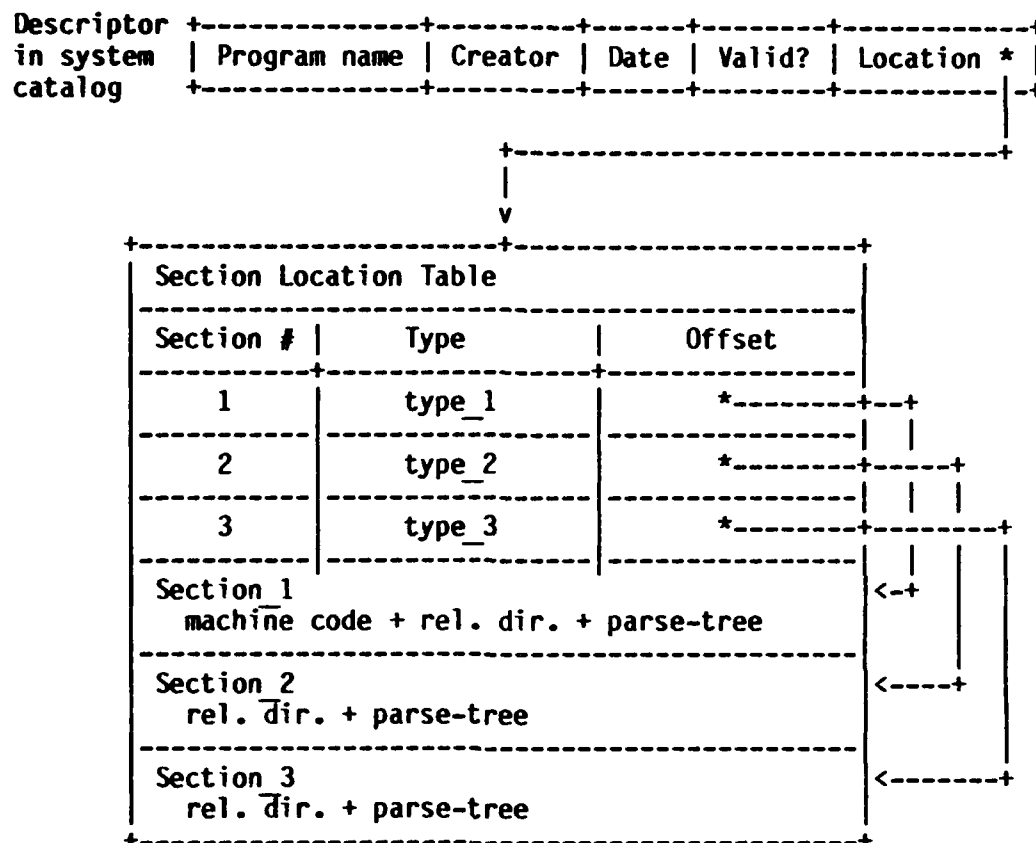


Figure 2.10 Structure of an Access Module

indexes it uses as access paths, and the privilege of its creator. Therefore, whenever a data object or index is dropped or a privilege is revoked, the system automatically performs a search in its internal catalogs to find access modules which are affected by the change. If the change involves dropping a data object or revoking a necessary privilege, the access module is erased from the database. However, if the change involves only dropping an index used by the access module, it will be possible to regenerate the access module by choosing an alternative access path. In this case, the access module is marked 'invalid'. When the access module is next invoked, the invalid marking is detected and the access module is regenerated automatically, based on the currently available access paths. The newly regenerated access module is stored in the database and also loaded into virtual memory for execution. The user is unaware of the regeneration process, except for a slight delay during the initial loading of the access module.

The user may try to change the database in such a way that would invalidate an access module while the access module is actually running. To prevent this from occurring, the 'transaction' mechanism is used. A programmer can declare transaction boundaries in his program by the BEGIN TRANSACTION and END TRANSACTION statements. The end of a transaction indicates that the database is in a consistent state. While a transaction is in progress, the loaded access module protects itself by holding a lock on its own description in the system catalog. Therefore, any database change that invalidates the access module must wait until the lock is released. At the end of each transaction, the running access module releases the lock on its own description, allowing any database changes which awaiting the lock to proceed. At the beginning of the next transaction, the access module tries to reacquire the lock on its own description. There are four possible outcomes (Chamberlin, Astrahan, King, et. al. 1981).

1. The description is still marked 'valid', and the timestamp in the description is unchanged. In this case, execution of the access module proceeds normally.
2. The description is gone. The access module has been destroyed by loss of an essential data object or privilege. An appropriate code is returned to the user's program, which is to continue.
3. The description is present but marked 'invalid'. This indicates that an index used by an access module is dropped. The access module is regenerated on the spot, choosing a new access path to replace the missing index. The user program then continues without interruption.
4. The description is marked 'valid', but its timestamp has changed (i.e. another user has caused a regeneration). The new access module is loaded into virtual memory and the user program continues.

For certain statements, no significant choice of access path is required. These statements include those which create and drop data objects and indexes, begin and end transactions, and grant and revoke privileges. The process of creating a data object, for example, involves placing its description in the system catalogs. Since, this process takes place essentially the same way for each new data object, it is possible to build standard routines for creating data objects. It is then unnecessary to

generate new machine code in an access module whenever a new object is created. Instead the standard program is invoked with necessary parameters.

A user may write a program that creates a temporary data object, processes it, then destroys it at the end of the run. When such program is precompiled, the optimizer is unable to choose an access path for processing such data object, because it does not yet exist. Whenever, the optimizer discovers that some object referenced in a statement does not exist, it places the parse-tree in a special section which indicates that the normal process of compilation has been terminated after the parsing step. At run-time RCS can not give control directly to this section, instead it makes another attempt to invoke the optimizer. This time, since the temporary data object is about to be operated on, it should be in existence. If the optimization is successful, the code generator is invoked and the machine language routine is generated and the version of the access module in virtual memory is updated. However, if the optimization fails because the indicated data object does not exist, a code is returned to the calling program indicating 'non-existent data object reference'.

Some programs execute statements that were not known at the time of precompilation. An example is Terminal Interface, which allows users to type data statements on ad hoc basis at a terminal and execute them and display the result. Such features are supported by PREPARE statement.

PREPARE <statement name> AS <string variable>

String variable contains a valid data statement. When the precompiler encounters a PREPARE statement, it creates a special zero-length section. At run-time RCS pass it through the parser, optimizer, and code generator. After the machine language code is generated the statement is ready for execution.

### B.3. Data Storage Interface

#### B.3.1 Area

The storage interface provides one potentially infinite linear address space. However, it is preferable to partition a large database into areas (Lorie 1977). Such a partition allows for smaller addresses to be used. It improves flexibility for controlling access to the database, and provides a means of factoring out some common attributes of a collection of data. It is also useful for selectively saving and restoring information.

The database consists of a set of disjoint areas, each of which constitutes a linear address space ( $A_k$ ,  $1 \leq k \leq N$ ). These areas are used for storing user data, access path structure, internal catalog information, and intermediate results. All the elements of a data object must reside within a single area; however, a given area may contain several data objects, indexes etc.

Areas are classified in three major types, depending on the combination of functions supported and overhead incurred.

1. Public area: They contain shared data that can be simultaneously accessed by multiple users.
2. Private area: They contain data that can be used by only one user at a time (or data that is not shared at all).
3. Temporary area: They contain only temporary data which is lost as soon as the program terminates.

Data in public and private area is recoverable (i.e. data will not be lost in the event of a failure), but not the one in temporary area. This reduces the overall overhead, as the overhead associated with full support of concurrent sharing needed for public data area, can be avoided for private and temporary data.

The addressing of a particular location in an area could be done at the byte level, by using a relative address from the beginning of the area. However, such a continuous space must be stored on auxiliary storage in records. We therefore, decompose the address space into logical pages, knowing that these pages will be stored on disk in physical slots of identical size.

An area  $A_k$  is therefore defined as an ordered set of pages of equal length. A page is referred to by its ordinal number in the segment, say  $i$ , with  $1 \leq i \leq M_k$ , where  $M_k$  is the maximum number of pages in area  $A_k$ .

#### B.3.2 Access Path

Each data object is represented as a stored file. Like an area, a stored file is identified at the DSI by a numeric identifier called DoID (Do for data object). The DLI is responsible for mapping user given object names to DoIDs.

Like areas and files, individual records have their own numeric identifier, called DaID (Da for data aggregate). They are used within the DSI to build indexes and links. The DaID access method to data object is a hybrid scheme, which combines the speed of a byte address pointer with the flexibility of indirection. Each DaID is a concatenation of a page number within the area, along with a byte offset from the bottom of the page. The offset denotes a special entry or 'slot' which contains the byte location of the data aggregate in that page. This technique allows efficient utilization of space within data pages, since space can be compacted and data aggregates moved with only local changes to the pointers in the slots. The slots themselves are never moved from their positions at the bottom of each data page, so the existing DaIDs can still be employed to access the data aggregates.

In the rare cases when a data aggregate is updated to a longer total value and insufficient space is available on its page, an overflow scheme is provided to move the data aggregate to another page. In this case the DaID points to a tagged overflow record, which is used to reference the other page. If it overflows again, the original overflow record is modified to point to the newest location. Therefore, access via a DaID almost always involves a single page access, and never involves more than two page accesses. Figure 3.1 shows how DaIDs are implemented.

In order to tune the database to particular environment, the DSI accepts hints for physical allocation during INSERT operations. These hints are in the form of a tentative DaID. The new data aggregate is inserted in the page associated with that DaID, if sufficient space is available. Otherwise, a nearby page is chosen by DSI. Use of this facility enables the DLI to cluster data aggregates with respect to some criteria such as value ordering on one or more fields. Another use is to cluster data aggregates of one object near particular data aggregate of another object, because of matching values in some of the fields. This clustering rule results in high performance for relational join operations, as well as for the support of hierarchical and network application.

## Images

An image is a logical ordering with respect to value in one or more sort fields. Images combined with scans provide the ability to scan data objects along a value ordering. Also, an image provides associative access capability. The DLI can rapidly fetch data aggregate from an image by keying on the sort field values. The DLI can also open a scan at a particular point in the image, and retrieve a sequence of data aggregates with a given range of sort values.

A new image can be defined at any time on any combination of fields; only restriction being that fields must be atomic. Furthermore, each of the fields may be specified as ascending or descending order. Once defined, an image is maintained automatically by the DSI. An image can also be dropped at any time.

The DSI maintains each image through the use of a multipage index structure (Bayer, McCreight 1972, Wagner 1973). An internal interface is used

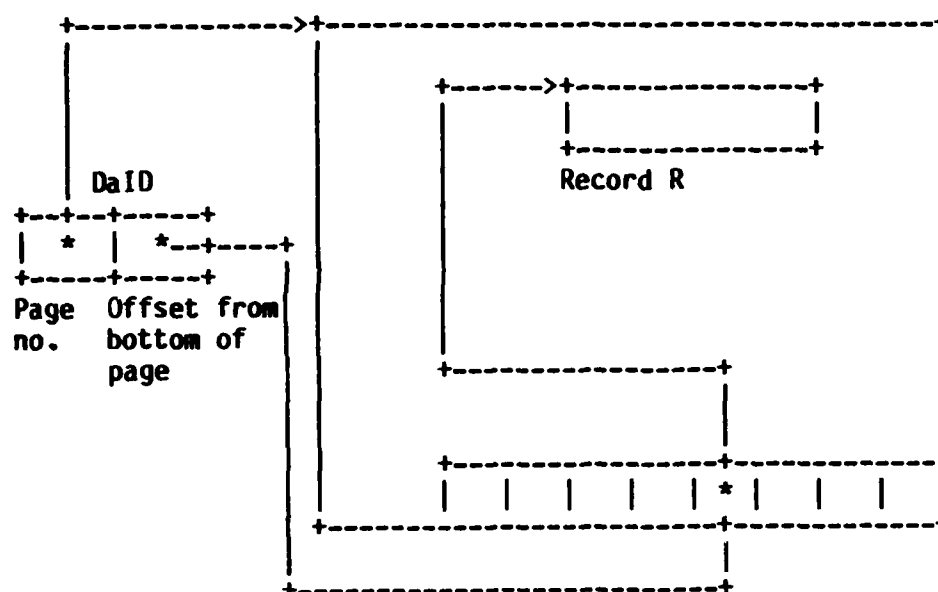


Figure 3.1 Implementation of DaID

for associative or sequential access along an image, and also to delete or insert index entries when data aggregates are deleted, inserted or updated. The parameters passed across this interface include the sort field values along with the DaID of the given data aggregate.

Each index is composed of one or more pages within the area containing the data object. A new page can be added to an index when needed as long as one of the pages within the area is marked available. The pages for a given index are organized into B-tree structure. Each page is a node and contains an ordered sequence of index entries. For nonleaf nodes, an entry consists of a <sort value, pointer> pair. The pointer addresses another page in the same structure, which may be either a leaf page or another nonleaf page. In either case the target page contains entries for sort values less than or equal to the given one. For the leaf nodes, an entry is a combination of sort values along with an ascending list of DaIDs for data aggregates having exactly those sort values. The leaf pages are chained in a doubly linked list, so that sequential access can be supported from leaf to leaf.

## Links

A link is an access path which is used to connect data aggregates in one or two data objects. The DLI decides which data aggregates will be on a link and determines their relative position. The DSI maintains internal pointers so that newly connected data aggregates are linked to previous and next twins; previous and next twins are linked to each other when a data aggregate is disconnected.

A unary link involves a single data object and provides a partially defined ordering of data aggregates. Unary links can be used to maintain ordering specification (not value ordered) of data aggregates, which are not supported by DSI. It also provides an efficient access path through all data aggregates of an object without the time overhead of an internal page scan.

The more important access path is a binary link. It provides a path from single data aggregates (parents) in one object to sequences of data aggregates (children) in another object. The DLI determines which data aggregates will be children under a given parent, and the relative order of children under a given parent. A data aggregate may be parents and/or children in an arbitrary number of different links. The only restriction is that a given data aggregate can appear only once within a given link.

The main use of binary links is to connect child data aggregates to a parent data aggregate, based on value matching in one or more fields. With such a structure the DLI can access data aggregates in one object based on the matching field in a data aggregate in a different object. This function is specially important for supporting relational join operations, and also for supporting navigational processing through hierarchical and network models of data. A striking advantage is gained over images when the children are clustered on the same page as the parent. Another important feature is that links provide reasonably fast associative access without the use of an extra index.

The links are maintained in the DSI by storing DaIDs in the prefix of data aggregates. New links can be defined at any time. When a new link is defined, a portion of the prefix is assigned to hold the required entries. An existing link can be dropped at any time. When this occurs, each data aggregate in the corresponding data object(s) is accessed by DSI, in order to invalidate the existing prefix entries and make space available for subsequent link definition.

### B.3.3 Concurrency Control

MIDAS is a concurrent user system. It employs locking techniques to solve various synchronization problem, both at the logical level of data objects and at the physical level of pages (Gray, Lorie, Putzolu 1975). If the transactions are not synchronized, the second update will overwrite the first, and the effect of one update will be lost. Similarly, a user may wish to read only clean data (in contrast to dirty data which have been updated by a transaction and which may be backed out). Also, if transaction recovery is to affect only the modifications of a single user, then mechanisms are needed to ensure that data updated by some ongoing transaction is not updated by another.

At the physical level of pages, locking technique is used to ensure that internal components of DSI give correct results. For example, a data page may contain several data aggregates, accessed by their identifiers, which require following a pointer within the data page. Even if no logical conflict occurs between two transactions, because each is accessing a different data object or different data aggregate of the same object, problems can occur at the physical level if one transaction follows a pointer to a data aggregate on some page while the other transaction updates a second data aggregate on the same page, and causes a data compaction routine to reassign data aggregate location.

Both logical and physical locking is handled by DSI. Physical locking is handled by setting and holding locks on one or more pages during the execution of a single DSI operation. Logical locking is handled by setting locks on such objects as areas, data objects, DaIDs and key value intervals, and holding them until they are explicitly released or to the end of the transaction. The level of locking in an area can be expanded to an entire page of data, rather than a single data aggregate. This allows pages to be locked for both logical and physical purposes, by varying the duration of the lock.

DSI employs a single lock mechanism to synchronize access to all objects. This synchronization is handled by a set of procedures in every activation of the DSI. It maintains a collection of queue structures called **gates**. Some of these gates are numbered and are associated with resources like table of buffer content. However, in order to handle locks on a potentially huge set of objects (like data aggregates), DSI includes named gate. Internal components can request a lock by giving a name for the object, using such names as DaID, index value, or page number. If the named resource is already locked, it will have a gate. The named gate will be deallocated when its queue becomes empty.

Locks are defined using several parameters such as shared, exclusive etc. If data elements are inserted or updated by a transaction, then an exclusive lock must be held on the data aggregate until the transaction has ended. If a data aggregate is deleted, then an exclusive lock must be held on the DaID of that data aggregate for the duration of the transaction. This guarantees that the deletion can be undone correctly during transaction backout. For any of these cases, an additional lock is set on the page itself to prevent conflict of transaction at the physical level; however, these page locks are released at the end of the DSI call.

Data items can be locked at various granularities. For example, locks on a single data aggregates are effective for transactions which access small amounts of data, while locks on entire data objects or even entire area are more reasonable for transactions which cause DLI to access large amounts of data. To accommodate these differences, a dynamic lock hierarchy protocol is used so that a small number of locks can be used to lock both few and many objects. This scheme essentially associates separate locks with each granularity of object.

Since locks are requested dynamically, it is possible for two or more concurrent activations of the DSI to deadlock. The DSI checks for deadlock situation by looking for cycles in a user-user matrix, every time a request is blocked. If a deadlock is detected one or more transactions are backed out. The selection of victim is based on the relative ages of transactions, and the duration of the locks. In general, the DSI selects the youngest transaction whose lock is of short duration. This transaction is then backed out to the save point preceding the offending lock request.

#### **B.3.4 Mapping**

There are several ways to map logical pages into physical slots. The easiest way is to allocate  $M_k$  contiguous slots to area  $A_k$  and mapping page  $j$  into the  $j$ th slot. This is advantageous if sequential processing is frequent on that area. In general, however, this simple static allocation scheme cannot be used, because it is impossible to foresee and preallocate space required for each area; also it leads to reservation of slots for yet unused pages.

A new scheme is used where limited contiguity of pages is maintained. A set of contiguous pages is first allocated; when more pages are needed, a new set (extent) of contiguous pages is allocated. No contiguity exists between different extents. The mapping information is organized as follows: If the entire disk volumes are used, then the mapping information can be kept as a small table containing a sequence of disk addresses. If portions of the volumes are used, then the table should also contain the extents. Ignoring the transition from one disk extent to another, we can consider the sequence of slots as being physically sequential. We partition the sequence of slots into sequences of  $n$  slots referred to as physical clusters in such a way that the seek time between two slots in the same cluster is small compared to the seek time between two slots in different clusters (e.g. a disk cylinder). We also partition the logical sequence of pages in an area into sequences of  $m$  logical pages referred to as logical clusters.

When the first slot allocation is done for a page in a logical cluster, the system associates a physical cluster with this logical cluster. Later on, any slot required for a page in the same logical cluster will be acquired from the same physical cluster, if possible. If a slot cannot be found in the right physical cluster, it will be allocated from a different physical cluster not yet allocated to a logical cluster.

For each area the mapping is implemented by using a vector  $V_k$  (page table) and MAP (bit map).  $V_k$  contains a sequence of slot numbers. The  $i$ th element indicates which slot is used to store the contents of page  $i$ . A null value indicates undefined page. MAP is defined as a vector of  $L$  bits (if there is a maximum of  $L$  slots available in the system) such that,

MAP ( $j$ ) = 1                      if slot  $j$  is busy.

MAP ( $j$ ) = 0                      if slot  $j$  is free.

The mapping technique is illustrated in Figure 3.2.

The size of  $V_k$  is one word per addressable page. The map is generally larger than the physical record size, and it is therefore splitted into blocks of equal size, each block covering a portion of  $V_k$ . Similarly, MAP is also splitted in a number of blocks.

Therefore, the database is stored by using the bit map MAP, the vectors  $V_k$  ( $1 \leq k \leq N$ ), and the contents of slots  $j$  for all  $j$  in a  $V_k$ . The consistency constraint, which is strictly maintained is that the  $j$ th bit in MAP is ON if  $j$  appears in some  $V_k$  and is OFF otherwise.

### B.3.5 Transaction Management

The main memory is divided into two main buffers; one pool of block buffer (BB) and one pool of page buffer (PB) (Lorie 1977). Accessing page  $i$  of area  $A_k$  implies,

1. FETCH  $V_k$  block covering  $V_k(i)$ ;
2. FIND  $j = V_k(i)$ ;
3. FETCH page stored in slot  $j$  in PB.

If the page is modified, the buffer in PB is flagged (MOD bit ON).

Sometimes the block is also modified. For example, when a new page  $i$  is defined. As the page was previously undefined,  $V_k(i)$  was null. The system finds an available slot, say  $j$ . A buffer is chosen in PB to contain page  $i$ . No fetching is needed as the page does not yet contain any meaningful information. The block has to be updated, so that  $V_k(i) = j$ . The block is also flagged in BB as modified (MOD bit ON).

When all buffer gets full and still more space is needed, the least recently used unmodified page is chosen for replacement. In case, all the pages are modified, the one least recently used among them is chosen. If the

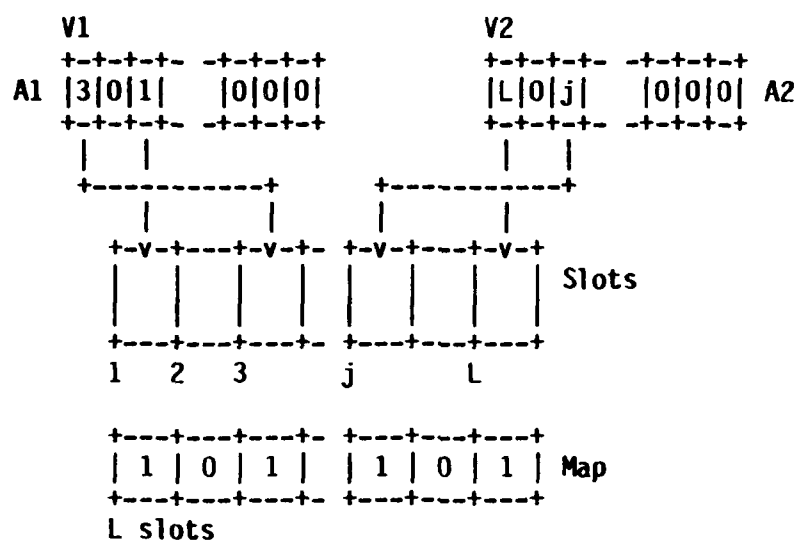


Figure 3.2 Mapping Between Pages and Slots

page was not modified, the buffer can be reused immediately; whereas if the page is modified, then its content has to be written back onto disk before it can be used.

When processing of area  $A_k$  is completed,  $V_k$  blocks and  $A_k$  pages are written back on disk if they are modified (MOD bit ON). This ensures that all modifications have been permanently stored in the database on disk.

### B.3.6 System Recovery

Let us suppose area  $A_k$  is in a state of integrity ( $D_k$ ) at time  $t$  and that a series of changes are made to  $A_k$  between time  $t$  and  $t + dt$ . At time  $t + dt$ , we store the new state  $D'_k$  onto disk. If the system failure occurs between  $t$  and  $t + dt$ , we should be able to return to the state  $D_k$ . If no failure occurs in that interval, the new state  $D'_k$  is recorded and  $D_k$  is forgotten. In case of subsequent failure one should be able to return to state  $D'_k$ . Therefore, we identify two basic requirements of a recovery feature.

1. One should be able to save the current state of the database as the new consistent state.
2. In case of failure, one should be able to return to the last state of integrity.

There may be two kinds of failure,

1. Hard failure: A disk volume may be physically damaged.
2. Soft failure: A hardware or software failure causes the contents of main memory to be lost, before the updates in buffer can be made permanent in database.

Transactions as described in previous section is highly vulnerable to a system failure. If the failure occurs after the area has been modified, the integrity of the area is lost. There may be two kinds of loss of integrity,

1. If some bit map or page table block has not yet been copied on disk, the mapping between pages and slots is lost, and therefore, the content of the database is lost.
2. If only some pages have not yet been written back on disk, the contents of these pages will be different from what the user expects.

#### B.3.6.1 Soft Failure

The essence of this method is to support immediate updates of the database (Lorie 1977) instead of deferring update till some later time (Severance, Lohman 1976). We retain original state  $D_k$  during the interval ( $t+dt$ ), and construct in parallel a new state  $D'_k$ . At time  $t + dt$ , we perform a single atomic operation (which cannot fail) to switch to  $D'_k$ .

Let us assume that at time  $t$ , the database is permanently stored on disk. For all  $k$  the state  $D_k$  is recorded in  $V_k$ , and the corresponding slots.

We define a second record for every block in  $V_k$  and third for MAP. We rename original page table as  $V_{k0}$  and the original bit map as  $MAP_0$ ; and name the alternate records  $V_{k1}$ , and  $MAP_1$  and  $MAP_2$ . We also define a record of  $N+1$  bits called master as follows :

MASTER : Record

STATUS : Array (1..N) of BIT(1);

MAP\_SWITCH : BIT(1)

End;

The MAP\_SWITCH indicates which of  $MAP_0$  or  $MAP_1$  is used. The STATUS bit indicates if the areas are open or closed. We assume that initially all the areas are closed and  $MAP_0$  contains the bit map,

i.e, MAP\_SWITCH = 0

and STATUS (h) = 0 ( $1 \leq h \leq N$ )

Such a situation is shown in Figure 3.3 (content of  $V_{11}$ ,  $V_{21}$ , and  $MAP_1$  is irrelevant).

While the areas are opened, following operations are performed.

1. Copy  $V_{k0}$  to  $V_{k1}$ .  $V_{k0}$  holds the current sets of values and  $V_{k1}$  is saved as backup.
2. Copy  $MAP_0$  to  $MAP_1$  and  $MAP_2$ .  $MAP_0$  is saved as backup and  $MAP_1$  is used for save operation.  $MAP_2$  holds the current version of the bit map.  $MAP_2$  may be stored same way as  $MAP_0$  and  $MAP_1$  on disk; however, if there is enough room in the main memory there is no need to copy it to the disk.
3. Update MASTER so that STATUS (K) = 1.

The last operation implies reading the master into the main memory, changing the  $k$ th STATUS bit, and rewriting the record. There is a possibility that the system crashes while the master is being rewritten. Such failure would be responsible for the loss of possibly the whole database. The solution is to have two copies of the master. Every write operation on the master triggers a second write if the first write is successful. If the first write fails, the second record still contains the previous master and the backup version is still available. Therefore, with this mechanism writing of master can never fail.

Suppose we want to modify page  $i$  of the area  $A_k$ . The entry  $j = V_{k0}(i)$  is found and the appropriate page is brought into PB and modified. Eventually, it will be swapped out and rewritten on disk. At that time it is not written back into its original slot but into a new free slot  $j'$ .  $V_{k0}$

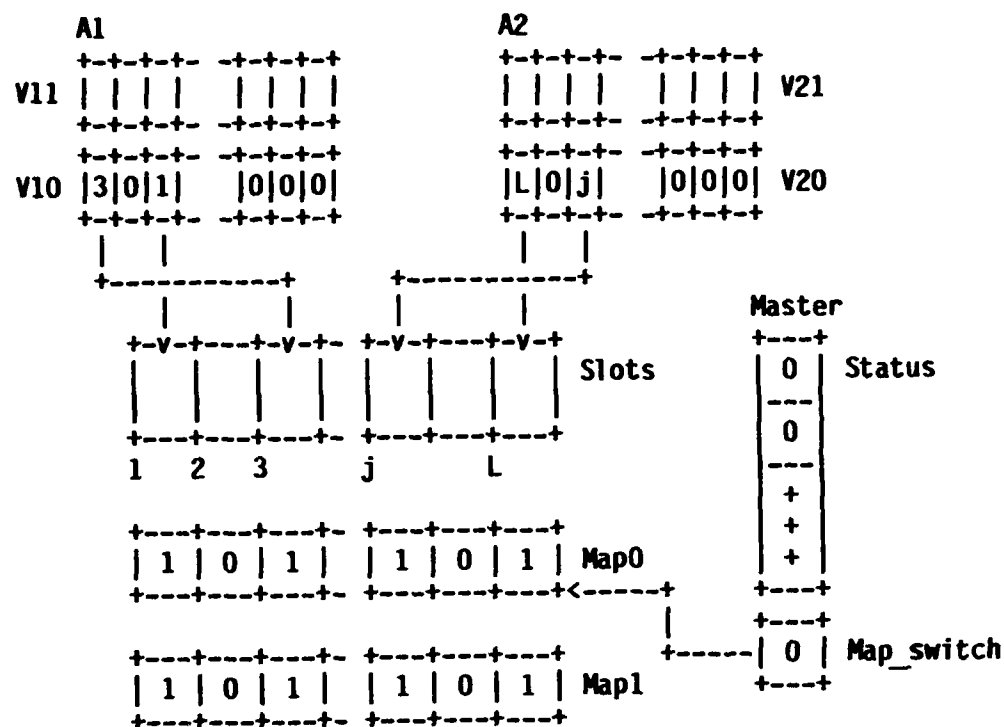


Figure 3.3 Database Representation (All Areas are Closed)

is updated so that  $Vk0(i) = j'$ . The old slot  $j$  is not released but kept as a shadow slot, pointed to by  $Vk1$ . Then  $Vk0(i)$  is flagged (shadow bit ON). Therefore,  $Vk0(i) \neq Vk1(i)$  if and only if the shadow bit for  $Vk0(i)$  is ON. Finding a free slot involves the use and modification of the bit map. MAP2 is used to find a free slot and mark it as busy.

Later, if the same page is modified again, it can be written back into the slot  $j'$  because the backup version of that page is stored in slot  $j$ . Figure 3.4 illustrates a possible situation after modification of A1 (content of V21 and MAP1 is irrelevant).

By interrogating shadow bits, we find out the values of  $i$  for which  $Vk0(i) \neq Vk1(i)$ . When this is true, MAP0 must be updated to reflect the fact that slot  $j' = Vk0(i)$  is now busy and slot  $j = Vk1(i)$  is now free. The current bit map already has  $j'$  busy,  $j$  can be freed. As the system could fail during the updating of MAP0, the new bit map value is instead developed into MAP1. When this operation is completed, the master is updated so that,

STATUS (k) = 0

and MAP\_SWITCH = 1

This indicates that the area is closed and MAP1 now contains the bit map. The content of MAP0 is now irrelevant.

Suppose there is a system failure (content of main memory is wiped out) before the current state could be saved in database. The system interrogates master if an area is open. If not it is in a state of integrity. If the area is open, the page table  $Vk1$  reflects the last state of integrity and can be recopied into  $Vk0$ . A system failure during the restore operation does not destroy any vital information. The procedure can just be reinitiated. Then the master is updated so that,

STATUS (h) = 0      (1 ≤ h ≤ N)

MAP\_SWITCH is unchanged and the integrity is restored.

To selectively restore one or more areas to their last state of integrity, we do as follows:

1. Use shadow bits to release slots in the current bit map.
2. Copy  $Vk1$  into  $Vk0$ .
3. Recopy the master with STATUS (k) = 0 .

Several areas can be restored or saved concurrently and the master can be updated in the end by one atomic operation. If one wants to resume processing on a saved or restored area, it must be opened again, but the system takes advantage of the fact that  $Vk0$  and  $Vk1$  are already equal and no copying is necessary.



### B.3.6.2 Hard Failure

The only protection against disk damage is to maintain two copies of the data on different volumes, i.e another disk or a tape. The probability of destroying both copies at the same time is typically small.

A method (Lorie 1977) consists in making a second copy of the changed pages only when a save is performed. But this increases seriously the time taken by the save operation. As the probability of damaging a disk is far smaller than the probability of a system failure, long term checkpoints (less frequent than save) are defined. Typical checkpoint cycle is illustrated in Figure 3.5. The length of the cycle can be dynamically changed.

Suppose, the state of the database at time 0 is saved on a tape T. This state is entirely defined by the bit map, the page table, and the contents of the used slots.

In this mechanism, we associate three flags with every  $V_k(i)$ , instead of one shadow bit. These flags are called the shadow bit, the cumulative shadow bit, and the long term shadow bit. Everytime a shadow bit is turned ON, the cumulative shadow bit is also turned ON. But although the shadow bits are turned off at every save, the cumulative shadow bits are not. When a long term checkpoint is taken the contents of the  $V_k$  which have atleast one cumulative bit ON are saved in an easily accessible work area. They are also copied onto T together with the bit map which has just been updated by the save. The cumulative shadow bits are then copied to long term shadow bits and turned off. From then on, an independent process p is activated which copies onto T the pages for which long term shadow bits are ON; p uses the copy of  $V_k$  in work area.

When the slots are being copied, the activity on the database can go on normally. Only restriction is that the slots being copied by p cannot be released by a following save before p completes. The first time a save wants to release a slot, it does not do so if the long term shadow bit is ON; instead it turns OFF the long term shadow bit. This is because any subsequent release of that page will refer to a slot which is not being copied during this checkpoint operation. When p completes, those slots which have been updated and saved since p started, are released.

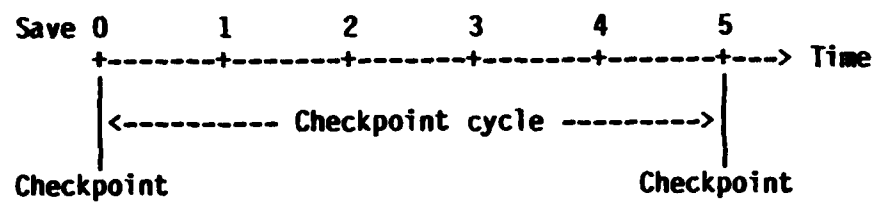


Figure 3.5 Save and Checkpoint Cycles

## REFERENCES

1. Astrahan, M.M., Blasgen, M.W., Chamberlin, D.D., et. al., (1976), "System R: Relational Approach to Database Management", ACM Trans. on Database Systems 1, 2 (June), pp: 97-137.
2. Bayer, R., McCreight, M., (1972), "Organization and Maintenance of Large Ordered Indexes", Acta Informatica 1, pp: 173-189.
3. Chamberlin, D.D. (1976), " Relational Database Management Systems", Computing Surveys 8, 1 (March), pp: 43-66.
4. Chamberlin, D.D., Astrahan, M.M., Eswaran, K.P., et. al., (1976), "SEQUEL 2: A Unified Approach to Data Definition, Manipulation and Control", IBM J. Res. Development (November), pp: 560-575.
5. Chamberlin, D.D., Astrahan, M.M., King, W.F., et. al., (1981), "Support for Repetitive Transactions and Ad Hoc Queries in System R", ACM Trans. on Database Systems 6, 1 (March), pp: 70-94.
6. Codd, E.F., (1970), "A Relational Model of Data for Large Shared Data Banks", Comm. ACM 13, 6 (June), pp: 377-387.
7. Comfort, D.L., Erickson, W.J., (1978), "RIM - A Prototype for a Relational Information Management System", NASA Conference Publications 2055. pp: 183-196.
8. Fry, J.P., Sibley, E.H., (1976), "Evolution of Database Management Systems", Computing Surveys 8, 1 (March), pp: 7-42.
9. Gray, J.N., Lorie, R.A., Putzolu, G.R., (1975), "Granularity of Locks in a Shared Database", Proc. Int. Conf. on Very Large Databases 1, 1 (September), pp: 428-451.
10. Griffiths, P.A., Wade, B.W., (1976), "An Authorization Mechanism for a Relational Database System", ACM Trans. on Database Systems 1,3 (September), pp: 242-255.
11. Lang, T., Fernandez, E.B., Summers, R.C., (1976), "A System Architecture for Compile-time Actions in Databases", IBM Los Angeles Scientific Centre, G320-2682 (December).
12. Lorie, R.A., (1977), "Physical Integrity in a Large Segmented Database", ACM Trans. on Database Systems 2, 1 (March), pp: 91-104.
13. Lorie, R.A., Wade, B.W., (1979), "The Compilation of a High Level Data Language", Research Report RJ2598, IBM Res. Lab., San Jose, Cal. 95193.
14. Lorie, R.A., Nilsson, J.F., (1979), "An Access Specification Language for a Relational Database System", IBM J. Res. Development 23, 3 (May), pp: 286-298.

15. Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., et. al., (1979), "Access Path Selection in a Relational Database Management System", Research Report RJ2429, IBM Res. Lab., San Jose, Cal. 95193.
16. Severance, D.G., Lohman, G.M., (1974) "Differential Files: Their Application to the Maintenance of Large Databases", ACM Trans. on Database Systems 1, 3 (September), pp: 256-267.
17. SreekantaMurthey, T., Reddy, C.P.D. and Arora, J.S., (1984), "Database Management Concepts in Engineering Design Optimization", Proceedings of the 26th AIAA/ASME/ASCE/AHS SDM Conference, Palm Springs, CA (May).
18. Stonebraker, M., Wong, E., Kreps, P., et. al., (1976), "The Design and Implementation of INGRESS", ACM Trans. on Database Systems 1, 3 (September), pp: 189-222.
19. Taylor, R.W., Frank, R.L., (1976), "CODASYL Database Management Systems", Computing Surveys 8, 1 (March), pp: 67-103.
20. Uhrowczik, P.P., (1973), "Data Dictionary/Directories", IBM System Journal 4, pp: 332-350.
21. Wagner, R.E., (1973), "Indexing Design Considerations", IBM Systems Journal 4, pp: 351-367.

**END**

**FILMED**

---

*1-86*

**DTIC**